NAME

intro — introduction to the system libraries

DESCRIPTION

This section provides an overview of the system libraries, their functions, error returns and other common definitions and concepts. Most of these functions are available from the standard C library, *libc*. Other libraries, such as the math library, *libm*, must be indicated at compile time with the **-l** option of the compiler.

The various system libraries supplied in NetBSD (followed by the linker flags) are:

libasn1 (**-l***asn1*)
: The abstract syntax notation (ASN) library provides routines for the specification of abstract data types.

libbz2 (**-l***bz2*)
: Block-sorting compressor library providing routines for fast and efficient compression.

libc (**-l***c*)
: The standard C library. When using the C compiler cc(1), it is not necessary to supply the linker flag **-l***c* for these functions. There are several subsystems included inside *libc*:

  standard I/O routines
  : see stdio(3)

  database routines
  : see db(3)

  bit operators
  : see bitstring(3)

  string operators
  : see string(3)

  character tests and character operators

  encryption and hash routines
  : see md4(3) and md5(3).

  storage allocation
  : see mpool(3) and malloc(3)

  time functions
  : see time(3)

  signal handling
  : see signal(3)

libcdk (**-l***cdk*)
: Curses development kit (CDK) library. See cdk(3).

libcom_err (**-l***com_err*)
: The common error description library. See com_err(3).

libcompat (**-l***compat*)
: Functions which are obsolete but are available for compatibility with 4.3BSD. In particular, a number of system call interfaces provided in previous releases of BSD have been included for source code compatibility. Use of these routines should, for the most part, be avoided. The manual page entry for each compatibility routine indicates the proper interface to use.

libcrypt (**-l***crypt*)
The crypt library. See crypt(3).

libcrypto (**-l***crypto*)
The OpenSSL cryptographic library. See crypto(3).

libcrypto_idea (**-l***crypto_idea*)
The OpenSSL cryptographic library routines for the IDEA algorithm. This algorithm is separated from libcrypto since the IDEA algorithm is protected by patents and its use is restricted.

libcrypto_rc5 (**-l***crypto_rc5*)
The OpenSSL cryptographic library routines for the RC5 algorithm. This algorithm is separated from libcrypto since the RC5 algorithm is protected by patents and its use is restricted.

libcurses (**-l***curses* **-l***termcap*)
Terminal independent screen management routines for two dimensional non-bitmap display terminals. See curses(3).

libdes (**-l***des*)
The OpenSSL cryptographic library for the DES algorithms. See des(3).

libedit (**-l***edit*)
The command-line editor or editline library. The editline library provides generic editing and history functions. See editline(3).

libform (**-l***form*)
The curses form library provides a terminal-independent form system using the curses library. The form library provides facilities for defining forms on terminals. See forms(3).

libgssapi (**-l***gssapi*)
The Generic Security Services (GSS) API library. This library provides verification services to applications and usually sits above the cryptographic libraries.

libhesiod (**-l***hesiod*)
The Hesiod library. This library provides routines for performing lookups of Hesiod information, which is stored as text records in the Domain Name Service. See hesiod(3).

libhdb (**-l***hdb*)
The Heimdal Kerberos 5 authentication/authorisation database access library.

libintl (**-l***intl*)
The internationalized message handling library. See gettext(3).

libipsec (**-l***ipsec*)
The IPsec policy control library. See ipsec_set_policy(3) and ipsec_strerror(3).

libkadm (**-l***kadm*)
The Kerberos IV administration server and client library.

libkadm5clnt (**-l***kadm5clnt*)
The Kerberos 5 administration client library.

libkadm5srv (**-l***kadm5srv*)
The Kerberos 5 administration server library.

libkafs (**-l***kafs*)
The Kerberos IV AFS library. See kafs(3).

libkdb (**-l***kdb*)
The Kerberos IV authentication/authorisation database access library.

libkrb (**-l***krb*)
The Kerberos IV library.

libkrb5 (**-l***krb5*)
The Kerberos 5 library. See krb5(3).

libkstream (**-l***kstream*)
Kerberos IV encrypted stream library.

libkvm (**-l***kvm*)
Kernel data access library. See kvm(3).

libl (**-l***l*) The library for lex(1).

libm (**-l***m*) The math library. See math(3).

libmenu (**-l***menu*)
The curses menu library. See menus(3).

libpcap (**-l***pcap*)
The packet capture library. See pcap(3).

libpci (**-l***pci*)
The PCI bus access library. See pci(3).

libposix (**-l***posix*)
The POSIX compatibility library provides a compatibility interface for POSIX functions which differ from the standard BSD interfaces. See chown(2) and rename(2).

libresolv (**-l***resolv*)
The DNS resolver library.

librmt (**-l***rmt*)
Remote magnetic tape library. See rmtops(3).

libroken (**-l***roken*)
A library containing compatibility functions used by Kerberos. It implements functionality required by the Kerberos implementation not implemented in the standard NetBSD libraries.

librpcsvc (**-l***rpcsvc*)
The Remote Procedure Call (RPC) services library. See rpc(3).

libskey (**-l***skey*)
The S/Key one-time password library. See skey(3).

libsl (**-l***sl*)

libss (**-l***ss*)

libssl (**-l***ssl*)
The secure sockets layer (SSL) library. See ssl(3).

libtelnet (**-l***telnet*)
The telnet library.

libtermcap (**-l***termcap*)
The terminal-independent operation library. See termcap(3).

libusb (**-l***usb*)
The Universal Serial Bus (USB) access library.

libutil (**-l***util*)
The system utilities library. See `util`(3).

libwrap (**-l***wrap*)
The TCP wrappers library. See `hosts_access`(3).

liby (**-l***y*) The library for `yacc`(1).

libz (**-l***z*) General-purpose compression library.

SEE ALSO

`cc`(1), `ld`(1), `nm`(1), `rtld`(1), `intro`(2)

HISTORY

An **intro** manual appeared in Version 7 AT&T UNIX.

NAME

ASN1_OBJECT_new, ASN1_OBJECT_free, − object allocation functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/asn1.h>

ASN1_OBJECT *ASN1_OBJECT_new(void);
void ASN1_OBJECT_free(ASN1_OBJECT *a);
```

DESCRIPTION

The ASN1_OBJECT allocation routines, allocate and free an ASN1_OBJECT structure, which represents an ASN1 OBJECT IDENTIFIER.

ASN1_OBJECT_new() allocates and initializes a ASN1_OBJECT structure.

ASN1_OBJECT_free() frees up the **ASN1_OBJECT** structure **a**.

NOTES

Although *ASN1_OBJECT_new()* allocates a new ASN1_OBJECT structure it is almost never used in applications. The ASN1 object utility functions such as *OBJ_nid2obj()* are used instead.

RETURN VALUES

If the allocation fails, *ASN1_OBJECT_new()* returns **NULL** and sets an error code that can be obtained by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

ASN1_OBJECT_free() returns no value.

SEE ALSO

ERR_get_error (3), *d2i_ASN1_OBJECT* (3)

HISTORY

ASN1_OBJECT_new() and *ASN1_OBJECT_free()* are available in all versions of SSLeay and OpenSSL.

NAME

ASN1_STRING_dup, ASN1_STRING_cmp, ASN1_STRING_set, ASN1_STRING_length, ASN1_STRING_length_set, ASN1_STRING_type, ASN1_STRING_data − ASN1_STRING utility functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/asn1.h>

int ASN1_STRING_length(ASN1_STRING *x);
unsigned char * ASN1_STRING_data(ASN1_STRING *x);

ASN1_STRING * ASN1_STRING_dup(ASN1_STRING *a);

int ASN1_STRING_cmp(ASN1_STRING *a, ASN1_STRING *b);

int ASN1_STRING_set(ASN1_STRING *str, const void *data, int len);

int ASN1_STRING_type(ASN1_STRING *x);

int ASN1_STRING_to_UTF8(unsigned char **out, ASN1_STRING *in);
```

DESCRIPTION

These functions allow an **ASN1_STRING** structure to be manipulated.

ASN1_STRING_length() returns the length of the content of **x**.

ASN1_STRING_data() returns an internal pointer to the data of **x**. Since this is an internal pointer it should **not** be freed or modified in any way.

ASN1_STRING_dup() returns a copy of the structure **a**.

ASN1_STRING_cmp() compares **a** and **b** returning 0 if the two are identical. The string types and content are compared.

ASN1_STRING_set() sets the data of string **str** to the buffer **data** or length **len**. The supplied data is copied. If **len** is −1 then the length is determined by strlen(data).

ASN1_STRING_type() returns the type of **x**, using standard constants such as **V_ASN1_OCTET_STRING**.

ASN1_STRING_to_UTF8() converts the string **in** to UTF8 format, the converted data is allocated in a buffer in ***out**. The length of **out** is returned or a negative error code. The buffer ***out** should be free using *OPENSSL_free()*.

NOTES

Almost all ASN1 types in OpenSSL are represented as an **ASN1_STRING** structure. Other types such as **ASN1_OCTET_STRING** are simply typedefed to **ASN1_STRING** and the functions call the **ASN1_STRING** equivalents. **ASN1_STRING** is also used for some **CHOICE** types which consist entirely of primitive string types such as **DirectoryString** and **Time**.

These functions should **not** be used to examine or modify **ASN1_INTEGER** or **ASN1_ENUMERATED** types: the relevant **INTEGER** or **ENUMERATED** utility functions should be used instead.

In general it cannot be assumed that the data returned by *ASN1_STRING_data()* is null terminated or does not contain embedded nulls. The actual format of the data will depend on the actual string type itself: for example for and IA5String the data will be ASCII, for a BMPString two bytes per character in big endian format, UTF8String will be in UTF8 format.

Similar care should be take to ensure the data is in the correct format when calling *ASN1_STRING_set()*.

RETURN VALUES

SEE ALSO

ERR_get_error (3)

HISTORY

NAME

ASN1_STRING_new, ASN1_STRING_type_new, ASN1_STRING_free − ASN1_STRING allocation functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/asn1.h>

ASN1_STRING * ASN1_STRING_new(void);
ASN1_STRING * ASN1_STRING_type_new(int type);
void ASN1_STRING_free(ASN1_STRING *a);
```

DESCRIPTION

ASN1_STRING_new() returns an allocated **ASN1_STRING** structure. Its type is undefined.

ASN1_STRING_type_new() returns an allocated **ASN1_STRING** structure of type **type**.

ASN1_STRING_free() frees up **a**.

NOTES

Other string types call the **ASN1_STRING** functions. For example *ASN1_OCTET_STRING_new()* calls ASN1_STRING_type(V_ASN1_OCTET_STRING).

RETURN VALUES

ASN1_STRING_new() and *ASN1_STRING_type_new()* return a valid ASN1_STRING structure or **NULL** if an error occurred.

ASN1_STRING_free() does not return a value.

SEE ALSO

ERR_get_error (3)

HISTORY

TBA

NAME

ASN1_STRING_print_ex, ASN1_STRING_print_ex_fp − ASN1_STRING output routines.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/asn1.h>

int ASN1_STRING_print_ex(BIO *out, ASN1_STRING *str, unsigned long flags);
int ASN1_STRING_print_ex_fp(FILE *fp, ASN1_STRING *str, unsigned long flags);
int ASN1_STRING_print(BIO *out, ASN1_STRING *str);
```

DESCRIPTION

These functions output an **ASN1_STRING** structure. **ASN1_STRING** is used to represent all the ASN1 string types.

ASN1_STRING_print_ex() outputs **str** to **out**, the format is determined by the options **flags**. *ASN1_STRING_print_ex_fp()* is identical except it outputs to **fp** instead.

ASN1_STRING_print() prints **str** to **out** but using a different format to *ASN1_STRING_print_ex()*. It replaces unprintable characters (other than CR, LF) with '.'.

NOTES

ASN1_STRING_print() is a legacy function which should be avoided in new applications.

Although there are a large number of options frequently **ASN1_STRFLGS_RFC2253** is suitable, or on UTF8 terminals **ASN1_STRFLGS_RFC2253 & ~ASN1_STRFLGS_ESC_MSB**.

The complete set of supported options for **flags** is listed below.

Various characters can be escaped. If **ASN1_STRFLGS_ESC_2253** is set the characters determined by RFC2253 are escaped. If **ASN1_STRFLGS_ESC_CTRL** is set control characters are escaped. If **ASN1_STRFLGS_ESC_MSB** is set characters with the MSB set are escaped: this option should **not** be used if the terminal correctly interprets UTF8 sequences.

Escaping takes several forms.

If the character being escaped is a 16 bit character then the form "\UXXXX" is used using exactly four characters for the hex representation. If it is 32 bits then "\WXXXXXXXX" is used using eight characters of its hex representation. These forms will only be used if UTF8 conversion is not set (see below).

Printable characters are normally escaped using the backslash '\' character. If **ASN1_STRFLGS_ESC_QUOTE** is set then the whole string is instead surrounded by double quote characters: this is arguably more readable than the backslash notation. Other characters use the "\XX" using exactly two characters of the hex representation.

If **ASN1_STRFLGS_UTF8_CONVERT** is set then characters are converted to UTF8 format first. If the terminal supports the display of UTF8 sequences then this option will correctly display multi byte characters.

If **ASN1_STRFLGS_IGNORE_TYPE** is set then the string type is not interpreted at all: everything is assumed to be one byte per character. This is primarily for debugging purposes and can result in confusing output in multi character strings.

If **ASN1_STRFLGS_SHOW_TYPE** is set then the string type itself is printed out before its value (for example "BMPSTRING"), this actually uses *ASN1_tag2str()*.

The content of a string instead of being interpreted can be "dumped": this just outputs the value of the string using the form #XXXX using hex format for each octet.

If **ASN1_STRFLGS_DUMP_ALL** is set then any type is dumped.

Normally non character string types (such as OCTET STRING) are assumed to be one byte per character, if **ASN1_STRFLGS_DUMP_UNKNOWN** is set then they will be dumped instead.

When a type is dumped normally just the content octets are printed, if **ASN1_STRFLGS_DUMP_DER** is set

then the complete encoding is dumped instead (including tag and length octets).

ASN1_STRFLGS_RFC2253 includes all the flags required by RFC2253. It is equivalent to:
ASN1_STRFLGS_ESC_2253 | ASN1_STRFLGS_ESC_CTRL | ASN1_STRFLGS_ESC_MSB | ASN1_STRFLGS_UTF8_CONVERT | ASN1_STRFLGS_DUMP_UNKNOWN ASN1_STRFLGS_DUMP_DER

SEE ALSO

X509_NAME_print_ex (3), *ASN1_tag2str* (3)

HISTORY

TBA

NAME

ASN1_generate_nconf, ASN1_generate_v3 − ASN1 generation functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/asn1.h>

ASN1_TYPE *ASN1_generate_nconf(char *str, CONF *nconf);
ASN1_TYPE *ASN1_generate_v3(char *str, X509V3_CTX *cnf);
```

DESCRIPTION

These functions generate the ASN1 encoding of a string in an **ASN1_TYPE** structure.

str contains the string to encode **nconf** or **cnf** contains the optional configuration information where additional strings will be read from. **nconf** will typically come from a config file wherease **cnf** is obtained from an **X509V3_CTX** structure which will typically be used by X509 v3 certificate extension functions. **cnf** or **nconf** can be set to **NULL** if no additional configuration will be used.

GENERATION STRING FORMAT

The actual data encoded is determined by the string **str** and the configuration information. The general format of the string is:

[modifier,]type[:value]

That is zero or more comma separated modifiers followed by a type followed by an optional colon and a value. The formats of **type**, **value** and **modifier** are explained below.

SUPPORTED TYPES

The supported types are listed below. Unless otherwise specified only the **ASCII** format is permissible.

BOOLEAN, **BOOL**
This encodes a boolean type. The **value** string is mandatory and should be **TRUE** or **FALSE**. Additionally **TRUE**, **true**, **Y**, **y**, **YES**, **yes**, **FALSE**, **false**, **N**, **n**, **NO** and **no** are acceptable.

NULL
Encode the **NULL** type, the **value** string must not be present.

INTEGER, **INT**
Encodes an ASN1 **INTEGER** type. The **value** string represents the value of the integer, it can be preceeded by a minus sign and is normally interpreted as a decimal value unless the prefix **0x** is included.

ENUMERATED, **ENUM**
Encodes the ASN1 **ENUMERATED** type, it is otherwise identical to **INTEGER**.

OBJECT, **OID**
Encodes an ASN1 **OBJECT IDENTIFIER**, the **value** string can be a short name, a long name or numerical format.

UTCTIME, **UTC**
Encodes an ASN1 **UTCTime** structure, the value should be in the format **YYMMDDHHMMSSZ**.

GENERALIZEDTIME, **GENTIME**
Encodes an ASN1 **GeneralizedTime** structure, the value should be in the format **YYYYMMDDHHMMSSZ**.

OCTETSTRING, **OCT**
Encodes an ASN1 **OCTET STRING**. **value** represents the contents of this structure, the format strings **ASCII** and **HEX** can be used to specify the format of **value**.

BITSTRING, **BITSTR**
Encodes an ASN1 **BIT STRING**. **value** represents the contents of this structure, the format strings **ASCII**, **HEX** and **BITLIST** can be used to specify the format of **value**.

If the format is anything other than **BITLIST** the number of unused bits is set to zero.

UNIVERSALSTRING, **UNIV**, **IA5**, **IA5STRING**, **UTF8**, **UTF8String**, **BMP**, **BMPSTRING**, **VISIBLESTRING**, **VISIBLE**, **PRINTABLESTRING**, **PRINTABLE**, **T61**, **T61STRING**, **TELETEXSTRING**, **GeneralString**, **NUMERICSTRING**, **NUMERIC**

These encode the corresponding string types. **value** represents the contents of this structure. The format can be **ASCII** or **UTF8**.

SEQUENCE, **SEQ**, **SET**

Formats the result as an ASN1 **SEQUENCE** or **SET** type. **value** should be a section name which will contain the contents. The field names in the section are ignored and the values are in the generated string format. If **value** is absent then an empty SEQUENCE will be encoded.

MODIFIERS

Modifiers affect the following structure, they can be used to add EXPLICIT or IMPLICIT tagging, add wrappers or to change the string format of the final type and value. The supported formats are documented below.

EXPLICIT, **EXP**

Add an explicit tag to the following structure. This string should be followed by a colon and the tag value to use as a decimal value.

By following the number with **U**, **A**, **P** or **C** UNIVERSAL, APPLICATION, PRIVATE or CONTEXT SPECIFIC tagging can be used, the default is CONTEXT SPECIFIC.

IMPLICIT, **IMP**

This is the same as **EXPLICIT** except IMPLICIT tagging is used instead.

OCTWRAP, **SEQWRAP**, **SETWRAP**, **BITWRAP**

The following structure is surrounded by an OCTET STRING, a SEQUENCE, a SET or a BIT STRING respectively. For a BIT STRING the number of unused bits is set to zero.

FORMAT

This specifies the format of the ultimate value. It should be followed by a colon and one of the strings **ASCII**, **UTF8**, **HEX** or **BITLIST**.

If no format specifier is included then **ASCII** is used. If **UTF8** is specified then the value string must be a valid **UTF8** string. For **HEX** the output must be a set of hex digits. **BITLIST** (which is only valid for a BIT STRING) is a comma separated list of the indices of the set bits, all other bits are zero.

EXAMPLES

A simple IA5String:

```
IA5STRING:Hello World
```

An IA5String explicitly tagged:

```
EXPLICIT:0,IA5STRING:Hello World
```

An IA5String explicitly tagged using APPLICATION tagging:

```
EXPLICIT:0A,IA5STRING:Hello World
```

A BITSTRING with bits 1 and 5 set and all others zero:

```
FORMAT=BITLIST,BITSTRING:1,5
```

A more complex example using a config file to produce a SEQUENCE consiting of a BOOL an OID and a UTF8String:

```
asn1 = SEQUENCE:seq_section

[seq_section]
```

```
field1 = BOOLEAN:TRUE
field2 = OID:commonName
field3 = UTF8:Third field
```

This example produces an RSAPrivateKey structure, this is the key contained in the file client.pem in all OpenSSL distributions (note: the field names such as 'coeff' are ignored and are present just for clarity):

```
asn1=SEQUENCE:private_key
[private_key]
version=INTEGER:0

n=INTEGER:0xBB6FE79432CC6EA2D8F970675A5A87BFBE1AFF0BE63E879F2AFFB93644\
D4D2C6D000430DEC66ABF47829E74B8C5108623A1C0EE8BE217B3AD8D36D5EB4FCA1D9

e=INTEGER:0x010001

d=INTEGER:0x6F05EAD2F27FFAEC84BEC360C4B928FD5F3A9865D0FCAAD291E2A52F4A\
F810DC6373278C006A0ABBA27DC8C63BF97F7E666E27C5284D7D3B1FFFE16B7A87B51D

p=INTEGER:0xF3929B9435608F8A22C208D86795271D54EBDFB09DDEF539AB083DA912\
D4BD57

q=INTEGER:0xC50016F89DFF2561347ED1186A46E150E28BF2D0F539A1594BBD7FE467\
46EC4F

exp1=INTEGER:0x9E7D4326C924AFC1DEA40B45650134966D6F9DFA3A7F9D698CD4ABEA\
9C0A39B9

exp2=INTEGER:0xBA84003BB95355AFB7C50DF140C60513D0BA51D637272E355E397779\
E7B2458F

coeff=INTEGER:0x30B9E4F2AFA5AC679F920FC83F1F2DF1BAF1779CF989447FABC2F5\
628657053A
```

This example is the corresponding public key in a SubjectPublicKeyInfo structure:

```
# Start with a SEQUENCE
asn1=SEQUENCE:pubkeyinfo

# pubkeyinfo contains an algorithm identifier and the public key wrapped
# in a BIT STRING
[pubkeyinfo]
algorithm=SEQUENCE:rsa_alg
pubkey=BITWRAP,SEQUENCE:rsapubkey

# algorithm ID for RSA is just an OID and a NULL
[rsa_alg]
algorithm=OID:rsaEncryption
parameter=NULL

# Actual public key: modulus and exponent
[rsapubkey]
n=INTEGER:0xBB6FE79432CC6EA2D8F970675A5A87BFBE1AFF0BE63E879F2AFFB93644\
D4D2C6D000430DEC66ABF47829E74B8C5108623A1C0EE8BE217B3AD8D36D5EB4FCA1D9

e=INTEGER:0x010001
```

RETURN VALUES

ASN1_generate_nconf() and *ASN1_generate_v3()* return the encoded data as an **ASN1_TYPE** structure or **NULL** if an error occurred.

The error codes that can be obtained by *ERR_get_error* (3).

SEE ALSO

ERR_get_error (3)

HISTORY

ASN1_generate_nconf() and *ASN1_generate_v3()* were added to OpenSSL 0.9.8

NAME

BIO_ctrl, BIO_callback_ctrl, BIO_ptr_ctrl, BIO_int_ctrl, BIO_reset, BIO_seek, BIO_tell, BIO_flush, BIO_eof, BIO_set_close, BIO_get_close, BIO_pending, BIO_wpending, BIO_ctrl_pending, BIO_ctrl_wpending, BIO_get_info_callback, BIO_set_info_callback − BIO control operations

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

long BIO_ctrl(BIO *bp,int cmd,long larg,void *parg);
long BIO_callback_ctrl(BIO *b, int cmd, void (*fp)(struct bio_st *, int, const char
char * BIO_ptr_ctrl(BIO *bp,int cmd,long larg);
long BIO_int_ctrl(BIO *bp,int cmd,long larg,int iarg);

int BIO_reset(BIO *b);
int BIO_seek(BIO *b, int ofs);
int BIO_tell(BIO *b);
int BIO_flush(BIO *b);
int BIO_eof(BIO *b);
int BIO_set_close(BIO *b,long flag);
int BIO_get_close(BIO *b);
int BIO_pending(BIO *b);
int BIO_wpending(BIO *b);
size_t BIO_ctrl_pending(BIO *b);
size_t BIO_ctrl_wpending(BIO *b);

int BIO_get_info_callback(BIO *b,bio_info_cb **cbp);
int BIO_set_info_callback(BIO *b,bio_info_cb *cb);

typedef void bio_info_cb(BIO *b, int oper, const char *ptr, int arg1, long arg2, lo
```

DESCRIPTION

BIO_ctrl(), *BIO_callback_ctrl()*, *BIO_ptr_ctrl()* and *BIO_int_ctrl()* are BIO ''control'' operations taking arguments of various types. These functions are not normally called directly, various macros are used instead. The standard macros are described below, macros specific to a particular type of BIO are described in the specific BIOs manual page as well as any special features of the standard calls.

BIO_reset() typically resets a BIO to some initial state, in the case of file related BIOs for example it rewinds the file pointer to the start of the file.

BIO_seek() resets a file related BIO's (that is file descriptor and FILE BIOs) file position pointer to **ofs** bytes from start of file.

BIO_tell() returns the current file position of a file related BIO.

BIO_flush() normally writes out any internally buffered data, in some cases it is used to signal EOF and that no more data will be written.

BIO_eof() returns 1 if the BIO has read EOF, the precise meaning of ''EOF'' varies according to the BIO type.

BIO_set_close() sets the BIO **b** close flag to **flag**. **flag** can take the value BIO_CLOSE or BIO_NOCLOSE. Typically BIO_CLOSE is used in a source/sink BIO to indicate that the underlying I/O stream should be closed when the BIO is freed.

BIO_get_close() returns the BIOs close flag.

BIO_pending(), *BIO_ctrl_pending()*, *BIO_wpending()* and *BIO_ctrl_wpending()* return the number of pending characters in the BIOs read and write buffers. Not all BIOs support these calls. *BIO_ctrl_pending()* and *BIO_ctrl_wpending()* return a size_t type and are functions, *BIO_pending()* and *BIO_wpending()* are macros which call *BIO_ctrl()*.

RETURN VALUES

BIO_reset() normally returns 1 for success and 0 or −1 for failure. File BIOs are an exception, they return 0 for success and −1 for failure.

BIO_seek() and *BIO_tell()* both return the current file position on success and −1 for failure, except file BIOs which for *BIO_seek()* always return 0 for success and −1 for failure.

BIO_flush() returns 1 for success and 0 or −1 for failure.

BIO_eof() returns 1 if EOF has been reached 0 otherwise.

BIO_set_close() always returns 1.

BIO_get_close() returns the close flag value: BIO_CLOSE or BIO_NOCLOSE.

BIO_pending(), *BIO_ctrl_pending()*, *BIO_wpending()* and *BIO_ctrl_wpending()* return the amount of pending data.

NOTES

BIO_flush(), because it can write data may return 0 or −1 indicating that the call should be retried later in a similar manner to *BIO_write()*. The *BIO_should_retry()* call should be used and appropriate action taken is the call fails.

The return values of *BIO_pending()* and *BIO_wpending()* may not reliably determine the amount of pending data in all cases. For example in the case of a file BIO some data may be available in the FILE structures internal buffers but it is not possible to determine this in a portably way. For other types of BIO they may not be supported.

Filter BIOs if they do not internally handle a particular *BIO_ctrl()* operation usually pass the operation to the next BIO in the chain. This often means there is no need to locate the required BIO for a particular operation, it can be called on a chain and it will be automatically passed to the relevant BIO. However this can cause unexpected results: for example no current filter BIOs implement *BIO_seek()*, but this may still succeed if the chain ends in a FILE or file descriptor BIO.

Source/sink BIOs return an 0 if they do not recognize the *BIO_ctrl()* operation.

BUGS

Some of the return values are ambiguous and care should be taken. In particular a return value of 0 can be returned if an operation is not supported, if an error occurred, if EOF has not been reached and in the case of *BIO_seek()* on a file BIO for a successful operation.

SEE ALSO

TBA

NAME

BIO_f_base64 − base64 BIO filter

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>
#include <openssl/evp.h>

BIO_METHOD *   BIO_f_base64(void);
```

DESCRIPTION

BIO_f_base64() returns the base64 BIO method. This is a filter BIO that base64 encodes any data written through it and decodes any data read through it.

Base64 BIOs do not support *BIO_gets()* or *BIO_puts()*.

BIO_flush() on a base64 BIO that is being written through is used to signal that no more data is to be encoded: this is used to flush the final block through the BIO.

The flag BIO_FLAGS_BASE64_NO_NL can be set with *BIO_set_flags()* to encode the data all on one line or expect the data to be all on one line.

NOTES

Because of the format of base64 encoding the end of the encoded block cannot always be reliably determined.

RETURN VALUES

BIO_f_base64() returns the base64 BIO method.

EXAMPLES

Base64 encode the string ‘‘Hello World\n’’ and write the result to standard output:

```
BIO *bio, *b64;
char message[] = "Hello World \n";

b64 = BIO_new(BIO_f_base64());
bio = BIO_new_fp(stdout, BIO_NOCLOSE);
bio = BIO_push(b64, bio);
BIO_write(bio, message, strlen(message));
BIO_flush(bio);

BIO_free_all(bio);
```

Read Base64 encoded data from standard input and write the decoded data to standard output:

```
BIO *bio, *b64, *bio_out;
char inbuf[512];
int inlen;

b64 = BIO_new(BIO_f_base64());
bio = BIO_new_fp(stdin, BIO_NOCLOSE);
bio_out = BIO_new_fp(stdout, BIO_NOCLOSE);
bio = BIO_push(b64, bio);
while((inlen = BIO_read(bio, inbuf, 512)) > 0)
       BIO_write(bio_out, inbuf, inlen);

BIO_free_all(bio);
```

BUGS

The ambiguity of EOF in base64 encoded data can cause additional data following the base64 encoded block to be misinterpreted.

There should be some way of specifying a test that the BIO can perform to reliably determine EOF (for example a MIME boundary).

SEE ALSO

TBA

NAME

BIO_f_buffer − buffering BIO

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD * BIO_f_buffer(void);

#define BIO_get_buffer_num_lines(b)    BIO_ctrl(b,BIO_C_GET_BUFF_NUM_LINES,0,NULL)
#define BIO_set_read_buffer_size(b,size) BIO_int_ctrl(b,BIO_C_SET_BUFF_SIZE,size,0)
#define BIO_set_write_buffer_size(b,size) BIO_int_ctrl(b,BIO_C_SET_BUFF_SIZE,size,1
#define BIO_set_buffer_size(b,size)    BIO_ctrl(b,BIO_C_SET_BUFF_SIZE,size,NULL)
#define BIO_set_buffer_read_data(b,buf,num) BIO_ctrl(b,BIO_C_SET_BUFF_READ_DATA,num
```

DESCRIPTION

BIO_f_buffer() returns the buffering BIO method.

Data written to a buffering BIO is buffered and periodically written to the next BIO in the chain. Data read from a buffering BIO comes from an internal buffer which is filled from the next BIO in the chain. Both *BIO_gets()* and *BIO_puts()* are supported.

Calling *BIO_reset()* on a buffering BIO clears any buffered data.

BIO_get_buffer_num_lines() returns the number of lines currently buffered.

BIO_set_read_buffer_size(), *BIO_set_write_buffer_size()* and *BIO_set_buffer_size()* set the read, write or both read and write buffer sizes to **size**. The initial buffer size is DEFAULT_BUFFER_SIZE, currently 1024. Any attempt to reduce the buffer size below DEFAULT_BUFFER_SIZE is ignored. Any buffered data is cleared when the buffer is resized.

BIO_set_buffer_read_data() clears the read buffer and fills it with **num** bytes of **buf**. If **num** is larger than the current buffer size the buffer is expanded.

NOTES

Buffering BIOs implement *BIO_gets()* by using *BIO_read()* operations on the next BIO in the chain. By prepending a buffering BIO to a chain it is therefore possible to provide *BIO_gets()* functionality if the following BIOs do not support it (for example SSL BIOs).

Data is only written to the next BIO in the chain when the write buffer fills or when *BIO_flush()* is called. It is therefore important to call *BIO_flush()* whenever any pending data should be written such as when removing a buffering BIO using *BIO_pop()*. *BIO_flush()* may need to be retried if the ultimate source/sink BIO is non blocking.

RETURN VALUES

BIO_f_buffer() returns the buffering BIO method.

BIO_get_buffer_num_lines() returns the number of lines buffered (may be 0).

BIO_set_read_buffer_size(), *BIO_set_write_buffer_size()* and *BIO_set_buffer_size()* return 1 if the buffer was successfully resized or 0 for failure.

BIO_set_buffer_read_data() returns 1 if the data was set correctly or 0 if there was an error.

SEE ALSO

TBA

NAME

BIO_f_cipher, BIO_set_cipher, BIO_get_cipher_status, BIO_get_cipher_ctx − cipher BIO filter

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>
#include <openssl/evp.h>

BIO_METHOD *   BIO_f_cipher(void);
void BIO_set_cipher(BIO *b,const EVP_CIPHER *cipher,
                unsigned char *key, unsigned char *iv, int enc);
int BIO_get_cipher_status(BIO *b)
int BIO_get_cipher_ctx(BIO *b, EVP_CIPHER_CTX **pctx)
```

DESCRIPTION

BIO_f_cipher() returns the cipher BIO method. This is a filter BIO that encrypts any data written through it, and decrypts any data read from it. It is a BIO wrapper for the cipher routines *EVP_CipherInit()*, *EVP_CipherUpdate()* and *EVP_CipherFinal()*.

Cipher BIOs do not support *BIO_gets()* or *BIO_puts()*.

BIO_flush() on an encryption BIO that is being written through is used to signal that no more data is to be encrypted: this is used to flush and possibly pad the final block through the BIO.

BIO_set_cipher() sets the cipher of BIO **b** to **cipher** using key **key** and IV **iv**. **enc** should be set to 1 for encryption and zero for decryption.

When reading from an encryption BIO the final block is automatically decrypted and checked when EOF is detected. *BIO_get_cipher_status()* is a *BIO_ctrl()* macro which can be called to determine whether the decryption operation was successful.

BIO_get_cipher_ctx() is a *BIO_ctrl()* macro which retrieves the internal BIO cipher context. The retrieved context can be used in conjunction with the standard cipher routines to set it up. This is useful when *BIO_set_cipher()* is not flexible enough for the applications needs.

NOTES

When encrypting *BIO_flush()* **must** be called to flush the final block through the BIO. If it is not then the final block will fail a subsequent decrypt.

When decrypting an error on the final block is signalled by a zero return value from the read operation. A successful decrypt followed by EOF will also return zero for the final read. *BIO_get_cipher_status()* should be called to determine if the decrypt was successful.

As always, if *BIO_gets()* or *BIO_puts()* support is needed then it can be achieved by preceding the cipher BIO with a buffering BIO.

RETURN VALUES

BIO_f_cipher() returns the cipher BIO method.

BIO_set_cipher() does not return a value.

BIO_get_cipher_status() returns 1 for a successful decrypt and 0 for failure.

BIO_get_cipher_ctx() currently always returns 1.

EXAMPLES

TBA

SEE ALSO

TBA

NAME

BIO_f_md, BIO_set_md, BIO_get_md, BIO_get_md_ctx − message digest BIO filter

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>
#include <openssl/evp.h>

BIO_METHOD *   BIO_f_md(void);
int BIO_set_md(BIO *b,EVP_MD *md);
int BIO_get_md(BIO *b,EVP_MD **mdp);
int BIO_get_md_ctx(BIO *b,EVP_MD_CTX **mdcp);
```

DESCRIPTION

BIO_f_md() returns the message digest BIO method. This is a filter BIO that digests any data passed through it, it is a BIO wrapper for the digest routines *EVP_DigestInit()*, *EVP_DigestUpdate()* and *EVP_DigestFinal()*.

Any data written or read through a digest BIO using *BIO_read()* and *BIO_write()* is digested.

BIO_gets(), if its **size** parameter is large enough finishes the digest calculation and returns the digest value. *BIO_puts()* is not supported.

BIO_reset() reinitialises a digest BIO.

BIO_set_md() sets the message digest of BIO **b** to **md**: this must be called to initialize a digest BIO before any data is passed through it. It is a *BIO_ctrl()* macro.

BIO_get_md() places the a pointer to the digest BIOs digest method in **mdp**, it is a *BIO_ctrl()* macro.

BIO_get_md_ctx() returns the digest BIOs context into **mdcp**.

NOTES

The context returned by *BIO_get_md_ctx()* can be used in calls to *EVP_DigestFinal()* and also the signature routines *EVP_SignFinal()* and *EVP_VerifyFinal()*.

The context returned by *BIO_get_md_ctx()* is an internal context structure. Changes made to this context will affect the digest BIO itself and the context pointer will become invalid when the digest BIO is freed.

After the digest has been retrieved from a digest BIO it must be reinitialized by calling *BIO_reset()*, or *BIO_set_md()* before any more data is passed through it.

If an application needs to call *BIO_gets()* or *BIO_puts()* through a chain containing digest BIOs then this can be done by prepending a buffering BIO.

Before OpenSSL 0.9.9 the call to *BIO_get_md_ctx()* would only work if the BIO had been initialized for example by calling *BIO_set_md()*). In OpenSSL 0.9.9 and later the context is always returned and the BIO is state is set to initialized. This allows applications to initialize the context externally if the standard calls such as *BIO_set_md()* are not sufficiently flexible.

RETURN VALUES

BIO_f_md() returns the digest BIO method.

BIO_set_md(), *BIO_get_md()* and *BIO_md_ctx()* return 1 for success and 0 for failure.

EXAMPLES

The following example creates a BIO chain containing an SHA1 and MD5 digest BIO and passes the string ‘‘Hello World’’ through it. Error checking has been omitted for clarity.

```
BIO *bio, *mdtmp;
char message[] = "Hello World";
bio = BIO_new(BIO_s_null());
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_sha1());
/* For BIO_push() we want to append the sink BIO and keep a note of
 * the start of the chain.
 */
bio = BIO_push(mdtmp, bio);
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_md5());
bio = BIO_push(mdtmp, bio);
/* Note: mdtmp can now be discarded */
BIO_write(bio, message, strlen(message));
```

The next example digests data by reading through a chain instead:

```
BIO *bio, *mdtmp;
char buf[1024];
int rdlen;
bio = BIO_new_file(file, "rb");
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_sha1());
bio = BIO_push(mdtmp, bio);
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_md5());
bio = BIO_push(mdtmp, bio);
do {
       rdlen = BIO_read(bio, buf, sizeof(buf));
       /* Might want to do something with the data here */
} while(rdlen > 0);
```

This next example retrieves the message digests from a BIO chain and outputs them. This could be used with the examples above.

```
BIO *mdtmp;
unsigned char mdbuf[EVP_MAX_MD_SIZE];
int mdlen;
int i;
mdtmp = bio;   /* Assume bio has previously been set up */
do {
       EVP_MD *md;
       mdtmp = BIO_find_type(mdtmp, BIO_TYPE_MD);
       if(!mdtmp) break;
       BIO_get_md(mdtmp, &md);
       printf("%s digest", OBJ_nid2sn(EVP_MD_type(md)));
       mdlen = BIO_gets(mdtmp, mdbuf, EVP_MAX_MD_SIZE);
       for(i = 0; i < mdlen; i++) printf(":%02X", mdbuf[i]);
       printf("\n");
       mdtmp = BIO_next(mdtmp);
} while(mdtmp);

BIO_free_all(bio);
```

BUGS

The lack of support for *BIO_puts()* and the non standard behaviour of *BIO_gets()* could be regarded as anomalous. It could be argued that *BIO_gets()* and *BIO_puts()* should be passed to the next BIO in the chain and digest the data passed through and that digests should be retrieved using a separate *BIO_ctrl()* call.

SEE ALSO

TBA

NAME

BIO_f_null − null filter

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *   BIO_f_null(void);
```

DESCRIPTION

BIO_f_null() returns the null filter BIO method. This is a filter BIO that does nothing.

All requests to a null filter BIO are passed through to the next BIO in the chain: this means that a BIO chain containing a null filter BIO behaves just as though the BIO was not there.

NOTES

As may be apparent a null filter BIO is not particularly useful.

RETURN VALUES

BIO_f_null() returns the null filter BIO method.

SEE ALSO

TBA

NAME

BIO_f_ssl, BIO_set_ssl, BIO_get_ssl, BIO_set_ssl_mode, BIO_set_ssl_renegotiate_bytes, BIO_get_num_renegotiates, BIO_set_ssl_renegotiate_timeout, BIO_new_ssl, BIO_new_ssl_connect, BIO_new_buffer_ssl_connect, BIO_ssl_copy_session_id, BIO_ssl_shutdown − SSL BIO

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>
#include <openssl/ssl.h>

BIO_METHOD *BIO_f_ssl(void);

#define BIO_set_ssl(b,ssl,c)   BIO_ctrl(b,BIO_C_SET_SSL,c,(char *)ssl)
#define BIO_get_ssl(b,sslp)    BIO_ctrl(b,BIO_C_GET_SSL,0,(char *)sslp)
#define BIO_set_ssl_mode(b,client)     BIO_ctrl(b,BIO_C_SSL_MODE,client,NULL)
#define BIO_set_ssl_renegotiate_bytes(b,num) \
       BIO_ctrl(b,BIO_C_SET_SSL_RENEGOTIATE_BYTES,num,NULL);
#define BIO_set_ssl_renegotiate_timeout(b,seconds) \
       BIO_ctrl(b,BIO_C_SET_SSL_RENEGOTIATE_TIMEOUT,seconds,NULL);
#define BIO_get_num_renegotiates(b) \
       BIO_ctrl(b,BIO_C_SET_SSL_NUM_RENEGOTIATES,0,NULL);

BIO *BIO_new_ssl(SSL_CTX *ctx,int client);
BIO *BIO_new_ssl_connect(SSL_CTX *ctx);
BIO *BIO_new_buffer_ssl_connect(SSL_CTX *ctx);
int BIO_ssl_copy_session_id(BIO *to,BIO *from);
void BIO_ssl_shutdown(BIO *bio);

#define BIO_do_handshake(b)    BIO_ctrl(b,BIO_C_DO_STATE_MACHINE,0,NULL)
```

DESCRIPTION

BIO_f_ssl() returns the SSL BIO method. This is a filter BIO which is a wrapper round the OpenSSL SSL routines adding a BIO ‘‘flavour’’ to SSL I/O.

I/O performed on an SSL BIO communicates using the SSL protocol with the SSLs read and write BIOs. If an SSL connection is not established then an attempt is made to establish one on the first I/O call.

If a BIO is appended to an SSL BIO using *BIO_push()* it is automatically used as the SSL BIOs read and write BIOs.

Calling *BIO_reset()* on an SSL BIO closes down any current SSL connection by calling *SSL_shutdown()*. *BIO_reset()* is then sent to the next BIO in the chain: this will typically disconnect the underlying transport. The SSL BIO is then reset to the initial accept or connect state.

If the close flag is set when an SSL BIO is freed then the internal SSL structure is also freed using *SSL_free()*.

BIO_set_ssl() sets the internal SSL pointer of BIO **b** to **ssl** using the close flag **c**.

BIO_get_ssl() retrieves the SSL pointer of BIO **b**, it can then be manipulated using the standard SSL library functions.

BIO_set_ssl_mode() sets the SSL BIO mode to **client**. If **client** is 1 client mode is set. If **client** is 0 server mode is set.

BIO_set_ssl_renegotiate_bytes() sets the renegotiate byte count to **num**. When set after every **num** bytes of I/O (read and write) the SSL session is automatically renegotiated. **num** must be at least 512 bytes.

BIO_set_ssl_renegotiate_timeout() sets the renegotiate timeout to **seconds**. When the renegotiate timeout elapses the session is automatically renegotiated.

BIO_get_num_renegotiates() returns the total number of session renegotiations due to I/O or timeout.

BIO_new_ssl() allocates an SSL BIO using SSL_CTX **ctx** and using client mode if **client** is non zero.

BIO_new_ssl_connect() creates a new BIO chain consisting of an SSL BIO (using **ctx**) followed by a connect BIO.

BIO_new_buffer_ssl_connect() creates a new BIO chain consisting of a buffering BIO, an SSL BIO (using **ctx**) and a connect BIO.

BIO_ssl_copy_session_id() copies an SSL session id between BIO chains **from** and **to**. It does this by locating the SSL BIOs in each chain and calling *SSL_copy_session_id()* on the internal SSL pointer.

BIO_ssl_shutdown() closes down an SSL connection on BIO chain **bio**. It does this by locating the SSL BIO in the chain and calling *SSL_shutdown()* on its internal SSL pointer.

BIO_do_handshake() attempts to complete an SSL handshake on the supplied BIO and establish the SSL connection. It returns 1 if the connection was established successfully. A zero or negative value is returned if the connection could not be established, the call *BIO_should_retry()* should be used for non blocking connect BIOs to determine if the call should be retried. If an SSL connection has already been established this call has no effect.

NOTES

SSL BIOs are exceptional in that if the underlying transport is non blocking they can still request a retry in exceptional circumstances. Specifically this will happen if a session renegotiation takes place during a *BIO_read()* operation, one case where this happens is when SGC or step up occurs.

In OpenSSL 0.9.6 and later the SSL flag SSL_AUTO_RETRY can be set to disable this behaviour. That is when this flag is set an SSL BIO using a blocking transport will never request a retry.

Since unknown *BIO_ctrl()* operations are sent through filter BIOs the servers name and port can be set using *BIO_set_host()* on the BIO returned by *BIO_new_ssl_connect()* without having to locate the connect BIO first.

Applications do not have to call *BIO_do_handshake()* but may wish to do so to separate the handshake process from other I/O processing.

RETURN VALUES

TBA

EXAMPLE

This SSL/TLS client example, attempts to retrieve a page from an SSL/TLS web server. The I/O routines are identical to those of the unencrypted example in *BIO_s_connect* (3).

```
BIO *sbio, *out;
int len;
char tmpbuf[1024];
SSL_CTX *ctx;
SSL *ssl;

ERR_load_crypto_strings();
ERR_load_SSL_strings();
OpenSSL_add_all_algorithms();

/* We would seed the PRNG here if the platform didn't
 * do it automatically
 */

ctx = SSL_CTX_new(SSLv23_client_method());

/* We'd normally set some stuff like the verify paths and
 * mode here because as things stand this will connect to
 * any server whose certificate is signed by any CA.
 */

sbio = BIO_new_ssl_connect(ctx);
```

```
BIO_get_ssl(sbio, &ssl);

if(!ssl) {
  fprintf(stderr, "Can't locate SSL pointer\n");
  /* whatever ... */
}

/* Don't want any retries */
SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);

/* We might want to do other things with ssl here */

BIO_set_conn_hostname(sbio, "localhost:https");

out = BIO_new_fp(stdout, BIO_NOCLOSE);
if(BIO_do_connect(sbio) <= 0) {
       fprintf(stderr, "Error connecting to server\n");
       ERR_print_errors_fp(stderr);
       /* whatever ... */
}

if(BIO_do_handshake(sbio) <= 0) {
       fprintf(stderr, "Error establishing SSL connection\n");
       ERR_print_errors_fp(stderr);
       /* whatever ... */
}

/* Could examine ssl here to get connection info */

BIO_puts(sbio, "GET / HTTP/1.0\n\n");
for(;;) {
       len = BIO_read(sbio, tmpbuf, 1024);
       if(len <= 0) break;
       BIO_write(out, tmpbuf, len);
}
BIO_free_all(sbio);
BIO_free(out);
```

Here is a simple server example. It makes use of a buffering BIO to allow lines to be read from the SSL BIO using BIO_gets. It creates a pseudo web page containing the actual request from a client and also echoes the request to standard output.

```
BIO *sbio, *bbio, *acpt, *out;
int len;
char tmpbuf[1024];
SSL_CTX *ctx;
SSL *ssl;

ERR_load_crypto_strings();
ERR_load_SSL_strings();
OpenSSL_add_all_algorithms();

/* Might seed PRNG here */

ctx = SSL_CTX_new(SSLv23_server_method());

if (!SSL_CTX_use_certificate_file(ctx,"server.pem",SSL_FILETYPE_PEM)
       || !SSL_CTX_use_PrivateKey_file(ctx,"server.pem",SSL_FILETYPE_PEM)
       || !SSL_CTX_check_private_key(ctx)) {
```

```
        fprintf(stderr, "Error setting up SSL_CTX\n");
        ERR_print_errors_fp(stderr);
        return 0;
}

/* Might do other things here like setting verify locations and
 * DH and/or RSA temporary key callbacks
 */

/* New SSL BIO setup as server */
sbio=BIO_new_ssl(ctx,0);

BIO_get_ssl(sbio, &ssl);

if(!ssl) {
  fprintf(stderr, "Can't locate SSL pointer\n");
  /* whatever ... */
}

/* Don't want any retries */
SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);

/* Create the buffering BIO */

bbio = BIO_new(BIO_f_buffer());

/* Add to chain */
sbio = BIO_push(bbio, sbio);

acpt=BIO_new_accept("4433");

/* By doing this when a new connection is established
 * we automatically have sbio inserted into it. The
 * BIO chain is now 'swallowed' by the accept BIO and
 * will be freed when the accept BIO is freed.
 */

BIO_set_accept_bios(acpt,sbio);

out = BIO_new_fp(stdout, BIO_NOCLOSE);

/* Setup accept BIO */
if(BIO_do_accept(acpt) <= 0) {
        fprintf(stderr, "Error setting up accept BIO\n");
        ERR_print_errors_fp(stderr);
        return 0;
}

/* Now wait for incoming connection */
if(BIO_do_accept(acpt) <= 0) {
        fprintf(stderr, "Error in connection\n");
        ERR_print_errors_fp(stderr);
        return 0;
}

/* We only want one connection so remove and free
 * accept BIO
 */

sbio = BIO_pop(acpt);

BIO_free_all(acpt);
```

```
if(BIO_do_handshake(sbio) <= 0) {
       fprintf(stderr, "Error in SSL handshake\n");
       ERR_print_errors_fp(stderr);
       return 0;
}
BIO_puts(sbio, "HTTP/1.0 200 OK\r\nContent-type: text/plain\r\n\r\n");
BIO_puts(sbio, "\r\nConnection Established\r\nRequest headers:\r\n");
BIO_puts(sbio, "--------------------------------------------------\r\n");
for(;;) {
       len = BIO_gets(sbio, tmpbuf, 1024);
       if(len <= 0) break;
       BIO_write(sbio, tmpbuf, len);
       BIO_write(out, tmpbuf, len);
       /* Look for blank line signifying end of headers*/
       if((tmpbuf[0] == '\r') || (tmpbuf[0] == '\n')) break;
}
BIO_puts(sbio, "--------------------------------------------------\r\n");
BIO_puts(sbio, "\r\n");
/* Since there is a buffering BIO present we had better flush it */
BIO_flush(sbio);
BIO_free_all(sbio);
```

SEE ALSO

TBA

NAME

BIO_find_type, BIO_next − BIO chain traversal

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO *  BIO_find_type(BIO *b,int bio_type);
BIO *  BIO_next(BIO *b);

#define BIO_method_type(b)             ((b)->method->type)

#define BIO_TYPE_NONE          0
#define BIO_TYPE_MEM           (1|0x0400)
#define BIO_TYPE_FILE          (2|0x0400)

#define BIO_TYPE_FD            (4|0x0400|0x0100)
#define BIO_TYPE_SOCKET                (5|0x0400|0x0100)
#define BIO_TYPE_NULL          (6|0x0400)
#define BIO_TYPE_SSL           (7|0x0200)
#define BIO_TYPE_MD            (8|0x0200)
#define BIO_TYPE_BUFFER                (9|0x0200)
#define BIO_TYPE_CIPHER                (10|0x0200)
#define BIO_TYPE_BASE64                (11|0x0200)
#define BIO_TYPE_CONNECT       (12|0x0400|0x0100)
#define BIO_TYPE_ACCEPT                (13|0x0400|0x0100)
#define BIO_TYPE_PROXY_CLIENT  (14|0x0200)
#define BIO_TYPE_PROXY_SERVER  (15|0x0200)
#define BIO_TYPE_NBIO_TEST     (16|0x0200)
#define BIO_TYPE_NULL_FILTER   (17|0x0200)
#define BIO_TYPE_BER           (18|0x0200)
#define BIO_TYPE_BIO           (19|0x0400)

#define BIO_TYPE_DESCRIPTOR    0x0100
#define BIO_TYPE_FILTER                0x0200
#define BIO_TYPE_SOURCE_SINK   0x0400
```

DESCRIPTION

The *BIO_find_type()* searches for a BIO of a given type in a chain, starting at BIO **b**. If **type** is a specific type (such as BIO_TYPE_MEM) then a search is made for a BIO of that type. If **type** is a general type (such as **BIO_TYPE_SOURCE_SINK**) then the next matching BIO of the given general type is searched for. *BIO_find_type()* returns the next matching BIO or NULL if none is found.

Note: not all the **BIO_TYPE_*** types above have corresponding BIO implementations.

BIO_next() returns the next BIO in a chain. It can be used to traverse all BIOs in a chain or used in conjunction with *BIO_find_type()* to find all BIOs of a certain type.

BIO_method_type() returns the type of a BIO.

RETURN VALUES

BIO_find_type() returns a matching BIO or NULL for no match.

BIO_next() returns the next BIO in a chain.

BIO_method_type() returns the type of the BIO **b**.

NOTES

BIO_next() was added to OpenSSL 0.9.6 to provide a 'clean' way to traverse a BIO chain or find multiple matches using *BIO_find_type()*. Previous versions had to use:

```
next = bio->next_bio;
```

BUGS

BIO_find_type() in OpenSSL 0.9.5a and earlier could not be safely passed a NULL pointer for the **b** argument.

EXAMPLE

Traverse a chain looking for digest BIOs:

```
BIO *btmp;
btmp = in_bio; /* in_bio is chain to search through */

do {
        btmp = BIO_find_type(btmp, BIO_TYPE_MD);
        if(btmp == NULL) break; /* Not found */
        /* btmp is a digest BIO, do something with it ...*/
        ...

        btmp = BIO_next(btmp);
} while(btmp);
```

SEE ALSO

TBA

NAME

BIO_new, BIO_set, BIO_free, BIO_vfree, BIO_free_all − BIO allocation and freeing functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO *  BIO_new(BIO_METHOD *type);
int    BIO_set(BIO *a,BIO_METHOD *type);
int    BIO_free(BIO *a);
void   BIO_vfree(BIO *a);
void   BIO_free_all(BIO *a);
```

DESCRIPTION

The *BIO_new()* function returns a new BIO using method **type**.

BIO_set() sets the method of an already existing BIO.

BIO_free() frees up a single BIO, *BIO_vfree()* also frees up a single BIO but it does not return a value. Calling *BIO_free()* may also have some effect on the underlying I/O structure, for example it may close the file being referred to under certain circumstances. For more details see the individual BIO_METHOD descriptions.

BIO_free_all() frees up an entire BIO chain, it does not halt if an error occurs freeing up an individual BIO in the chain.

RETURN VALUES

BIO_new() returns a newly created BIO or NULL if the call fails.

BIO_set(), *BIO_free()* return 1 for success and 0 for failure.

BIO_free_all() and *BIO_vfree()* do not return values.

NOTES

Some BIOs (such as memory BIOs) can be used immediately after calling *BIO_new()*. Others (such as file BIOs) need some additional initialization, and frequently a utility function exists to create and initialize such BIOs.

If *BIO_free()* is called on a BIO chain it will only free one BIO resulting in a memory leak.

Calling *BIO_free_all()* a single BIO has the same effect as calling *BIO_free()* on it other than the discarded return value.

Normally the **type** argument is supplied by a function which returns a pointer to a BIO_METHOD. There is a naming convention for such functions: a source/sink BIO is normally called BIO_s_*() and a filter BIO BIO_f_*();

EXAMPLE

Create a memory BIO:

```
BIO *mem = BIO_new(BIO_s_mem());
```

SEE ALSO

TBA

NAME

BIO_push, BIO_pop − add and remove BIOs from a chain.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO *  BIO_push(BIO *b,BIO *append);
BIO *  BIO_pop(BIO *b);
```

DESCRIPTION

The *BIO_push()* function appends the BIO **append** to **b**, it returns **b**.

BIO_pop() removes the BIO **b** from a chain and returns the next BIO in the chain, or NULL if there is no next BIO. The removed BIO then becomes a single BIO with no association with the original chain, it can thus be freed or attached to a different chain.

NOTES

The names of these functions are perhaps a little misleading. *BIO_push()* joins two BIO chains whereas *BIO_pop()* deletes a single BIO from a chain, the deleted BIO does not need to be at the end of a chain.

The process of calling *BIO_push()* and *BIO_pop()* on a BIO may have additional consequences (a control call is made to the affected BIOs) any effects will be noted in the descriptions of individual BIOs.

EXAMPLES

For these examples suppose **md1** and **md2** are digest BIOs, **b64** is a base64 BIO and **f** is a file BIO.

If the call:

```
BIO_push(b64, f);
```

is made then the new chain will be **b64−chain**. After making the calls

```
BIO_push(md2, b64);
BIO_push(md1, md2);
```

the new chain is **md1−md2−b64−f**. Data written to **md1** will be digested by **md1** and **md2**, **base64** encoded and written to **f**.

It should be noted that reading causes data to pass in the reverse direction, that is data is read from **f**, base64 **decoded** and digested by **md1** and **md2**. If the call:

```
BIO_pop(md2);
```

The call will return **b64** and the new chain will be **md1−b64−f** data can be written to **md1** as before.

RETURN VALUES

BIO_push() returns the end of the chain, **b**.

BIO_pop() returns the next BIO in the chain, or NULL if there is no next BIO.

SEE ALSO

TBA

NAME

BIO_read, BIO_write, BIO_gets, BIO_puts − BIO I/O functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

int    BIO_read(BIO *b, void *buf, int len);
int    BIO_gets(BIO *b,char *buf, int size);
int    BIO_write(BIO *b, const void *buf, int len);
int    BIO_puts(BIO *b,const char *buf);
```

DESCRIPTION

BIO_read() attempts to read **len** bytes from BIO **b** and places the data in **buf**.

BIO_gets() performs the BIOs ''gets'' operation and places the data in **buf**. Usually this operation will attempt to read a line of data from the BIO of maximum length **len**. There are exceptions to this however, for example *BIO_gets()* on a digest BIO will calculate and return the digest and other BIOs may not support *BIO_gets()* at all.

BIO_write() attempts to write **len** bytes from **buf** to BIO **b**.

BIO_puts() attempts to write a null terminated string **buf** to BIO **b**

RETURN VALUES

All these functions return either the amount of data successfully read or written (if the return value is positive) or that no data was successfully read or written if the result is 0 or −1. If the return value is −2 then the operation is not implemented in the specific BIO type.

NOTES

A 0 or −1 return is not necessarily an indication of an error. In particular when the source/sink is non-blocking or of a certain type it may merely be an indication that no data is currently available and that the application should retry the operation later.

One technique sometimes used with blocking sockets is to use a system call (such as *select()*, *poll()* or equivalent) to determine when data is available and then call *read()* to read the data. The equivalent with BIOs (that is call *select()* on the underlying I/O structure and then call *BIO_read()* to read the data) should **not** be used because a single call to *BIO_read()* can cause several reads (and writes in the case of SSL BIOs) on the underlying I/O structure and may block as a result. Instead *select()* (or equivalent) should be combined with non blocking I/O so successive reads will request a retry instead of blocking.

See *BIO_should_retry* (3) for details of how to determine the cause of a retry and other I/O issues.

If the *BIO_gets()* function is not supported by a BIO then it possible to work around this by adding a buffering BIO *BIO_f_buffer* (3) to the chain.

SEE ALSO

BIO_should_retry (3)

TBA

NAME

BIO_s_accept, BIO_set_accept_port, BIO_get_accept_port, BIO_set_nbio_accept, BIO_set_accept_bios, BIO_set_bind_mode, BIO_get_bind_mode, BIO_do_accept − accept BIO

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *BIO_s_accept(void);

long BIO_set_accept_port(BIO *b, char *name);
char *BIO_get_accept_port(BIO *b);

BIO *BIO_new_accept(char *host_port);

long BIO_set_nbio_accept(BIO *b, int n);
long BIO_set_accept_bios(BIO *b, char *bio);

long BIO_set_bind_mode(BIO *b, long mode);
long BIO_get_bind_mode(BIO *b, long dummy);

#define BIO_BIND_NORMAL                 0
#define BIO_BIND_REUSEADDR_IF_UNUSED    1
#define BIO_BIND_REUSEADDR              2

int BIO_do_accept(BIO *b);
```

DESCRIPTION

BIO_s_accept() returns the accept BIO method. This is a wrapper round the platform's TCP/IP socket accept routines.

Using accept BIOs, TCP/IP connections can be accepted and data transferred using only BIO routines. In this way any platform specific operations are hidden by the BIO abstraction.

Read and write operations on an accept BIO will perform I/O on the underlying connection. If no connection is established and the port (see below) is set up properly then the BIO waits for an incoming connection.

Accept BIOs support *BIO_puts()* but not *BIO_gets()*.

If the close flag is set on an accept BIO then any active connection on that chain is shutdown and the socket closed when the BIO is freed.

Calling *BIO_reset()* on a accept BIO will close any active connection and reset the BIO into a state where it awaits another incoming connection.

BIO_get_fd() and *BIO_set_fd()* can be called to retrieve or set the accept socket. See *BIO_s_fd* (3)

BIO_set_accept_port() uses the string **name** to set the accept port. The port is represented as a string of the form ''host:port'', where ''host'' is the interface to use and ''port'' is the port. Either or both values can be ''*'' which is interpreted as meaning any interface or port respectively. ''port'' has the same syntax as the port specified in *BIO_set_conn_port()* for connect BIOs, that is it can be a numerical port string or a string to lookup using *getservbyname()* and a string table.

BIO_new_accept() combines *BIO_new()* and *BIO_set_accept_port()* into a single call: that is it creates a new accept BIO with port **host_port**.

BIO_set_nbio_accept() sets the accept socket to blocking mode (the default) if **n** is 0 or non blocking mode if **n** is 1.

BIO_set_accept_bios() can be used to set a chain of BIOs which will be duplicated and prepended to the chain when an incoming connection is received. This is useful if, for example, a buffering or SSL BIO is required for each connection. The chain of BIOs must not be freed after this call, they will be automatically freed when the accept BIO is freed.

BIO_set_bind_mode() and *BIO_get_bind_mode()* set and retrieve the current bind mode. If BIO_BIND_NORMAL (the default) is set then another socket cannot be bound to the same port. If BIO_BIND_REUSEADDR is set then other sockets can bind to the same port. If BIO_BIND_REUSEADDR_IF_UNUSED is set then and attempt is first made to use BIO_BIN_NORMAL, if this fails and the port is not in use then a second attempt is made using BIO_BIND_REUSEADDR.

BIO_do_accept() serves two functions. When it is first called, after the accept BIO has been setup, it will attempt to create the accept socket and bind an address to it. Second and subsequent calls to *BIO_do_accept()* will await an incoming connection, or request a retry in non blocking mode.

NOTES

When an accept BIO is at the end of a chain it will await an incoming connection before processing I/O calls. When an accept BIO is not at then end of a chain it passes I/O calls to the next BIO in the chain.

When a connection is established a new socket BIO is created for the connection and appended to the chain. That is the chain is now accept−>socket. This effectively means that attempting I/O on an initial accept socket will await an incoming connection then perform I/O on it.

If any additional BIOs have been set using *BIO_set_accept_bios()* then they are placed between the socket and the accept BIO, that is the chain will be accept−>otherbios−>socket.

If a server wishes to process multiple connections (as is normally the case) then the accept BIO must be made available for further incoming connections. This can be done by waiting for a connection and then calling:

```
connection = BIO_pop(accept);
```

After this call **connection** will contain a BIO for the recently established connection and **accept** will now be a single BIO again which can be used to await further incoming connections. If no further connections will be accepted the **accept** can be freed using *BIO_free()*.

If only a single connection will be processed it is possible to perform I/O using the accept BIO itself. This is often undesirable however because the accept BIO will still accept additional incoming connections. This can be resolved by using *BIO_pop()* (see above) and freeing up the accept BIO after the initial connection.

If the underlying accept socket is non-blocking and *BIO_do_accept()* is called to await an incoming connection it is possible for *BIO_should_io_special()* with the reason BIO_RR_ACCEPT. If this happens then it is an indication that an accept attempt would block: the application should take appropriate action to wait until the underlying socket has accepted a connection and retry the call.

BIO_set_accept_port(), *BIO_get_accept_port()*, *BIO_set_nbio_accept()*, *BIO_set_accept_bios()*, *BIO_set_bind_mode()*, *BIO_get_bind_mode()* and *BIO_do_accept()* are macros.

RETURN VALUES

TBA

EXAMPLE

This example accepts two connections on port 4444, sends messages down each and finally closes both down.

```
BIO *abio, *cbio, *cbio2;
ERR_load_crypto_strings();
abio = BIO_new_accept("4444");

/* First call to BIO_accept() sets up accept BIO */
if(BIO_do_accept(abio) <= 0) {
       fprintf(stderr, "Error setting up accept\n");
       ERR_print_errors_fp(stderr);
       exit(0);
}
```

```
/* Wait for incoming connection */
if(BIO_do_accept(abio) <= 0) {
       fprintf(stderr, "Error accepting connection\n");
       ERR_print_errors_fp(stderr);
       exit(0);
}
fprintf(stderr, "Connection 1 established\n");
/* Retrieve BIO for connection */
cbio = BIO_pop(abio);
BIO_puts(cbio, "Connection 1: Sending out Data on initial connection\n");
fprintf(stderr, "Sent out data on connection 1\n");
/* Wait for another connection */
if(BIO_do_accept(abio) <= 0) {
       fprintf(stderr, "Error accepting connection\n");
       ERR_print_errors_fp(stderr);
       exit(0);
}
fprintf(stderr, "Connection 2 established\n");
/* Close accept BIO to refuse further connections */
cbio2 = BIO_pop(abio);
BIO_free(abio);
BIO_puts(cbio2, "Connection 2: Sending out Data on second\n");
fprintf(stderr, "Sent out data on connection 2\n");

BIO_puts(cbio, "Connection 1: Second connection established\n");
/* Close the two established connections */
BIO_free(cbio);
BIO_free(cbio2);
```

SEE ALSO

TBA

NAME

BIO_s_bio, BIO_make_bio_pair, BIO_destroy_bio_pair, BIO_shutdown_wr, BIO_set_write_buf_size, BIO_get_write_buf_size, BIO_new_bio_pair, BIO_get_write_guarantee, BIO_ctrl_get_write_guarantee, BIO_get_read_request, BIO_ctrl_get_read_request, BIO_ctrl_reset_read_request − BIO pair BIO

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *BIO_s_bio(void);

#define BIO_make_bio_pair(b1,b2)   (int)BIO_ctrl(b1,BIO_C_MAKE_BIO_PAIR,0,b2)
#define BIO_destroy_bio_pair(b)    (int)BIO_ctrl(b,BIO_C_DESTROY_BIO_PAIR,0,NULL)

#define BIO_shutdown_wr(b) (int)BIO_ctrl(b, BIO_C_SHUTDOWN_WR, 0, NULL)

#define BIO_set_write_buf_size(b,size) (int)BIO_ctrl(b,BIO_C_SET_WRITE_BUF_SIZE,siz
#define BIO_get_write_buf_size(b,size) (size_t)BIO_ctrl(b,BIO_C_GET_WRITE_BUF_SIZE,

int BIO_new_bio_pair(BIO **bio1, size_t writebuf1, BIO **bio2, size_t writebuf2);

#define BIO_get_write_guarantee(b) (int)BIO_ctrl(b,BIO_C_GET_WRITE_GUARANTEE,0,NULL
size_t BIO_ctrl_get_write_guarantee(BIO *b);

#define BIO_get_read_request(b)    (int)BIO_ctrl(b,BIO_C_GET_READ_REQUEST,0,NULL)
size_t BIO_ctrl_get_read_request(BIO *b);

int BIO_ctrl_reset_read_request(BIO *b);
```

DESCRIPTION

BIO_s_bio() returns the method for a BIO pair. A BIO pair is a pair of source/sink BIOs where data written to either half of the pair is buffered and can be read from the other half. Both halves must usually by handled by the same application thread since no locking is done on the internal data structures.

Since BIO chains typically end in a source/sink BIO it is possible to make this one half of a BIO pair and have all the data processed by the chain under application control.

One typical use of BIO pairs is to place TLS/SSL I/O under application control, this can be used when the application wishes to use a non standard transport for TLS/SSL or the normal socket routines are inappropriate.

Calls to *BIO_read()* will read data from the buffer or request a retry if no data is available.

Calls to *BIO_write()* will place data in the buffer or request a retry if the buffer is full.

The standard calls *BIO_ctrl_pending()* and *BIO_ctrl_wpending()* can be used to determine the amount of pending data in the read or write buffer.

BIO_reset() clears any data in the write buffer.

BIO_make_bio_pair() joins two separate BIOs into a connected pair.

BIO_destroy_pair() destroys the association between two connected BIOs. Freeing up any half of the pair will automatically destroy the association.

BIO_shutdown_wr() is used to close down a BIO **b**. After this call no further writes on BIO **b** are allowed (they will return an error). Reads on the other half of the pair will return any pending data or EOF when all pending data has been read.

BIO_set_write_buf_size() sets the write buffer size of BIO **b** to **size**. If the size is not initialized a default value is used. This is currently 17K, sufficient for a maximum size TLS record.

BIO_get_write_buf_size() returns the size of the write buffer.

BIO_new_bio_pair() combines the calls to *BIO_new()*, *BIO_make_bio_pair()* and *BIO_set_write_buf_size()* to create a connected pair of BIOs **bio1**, **bio2** with write buffer sizes **writebuf1**

and **writebuf2**. If either size is zero then the default size is used. *BIO_new_bio_pair()* does not check whether **bio1** or **bio2** do point to some other BIO, the values are overwritten, *BIO_free()* is not called.

BIO_get_write_guarantee() and *BIO_ctrl_get_write_guarantee()* return the maximum length of data that can be currently written to the BIO. Writes larger than this value will return a value from *BIO_write()* less than the amount requested or if the buffer is full request a retry. *BIO_ctrl_get_write_guarantee()* is a function whereas *BIO_get_write_guarantee()* is a macro.

BIO_get_read_request() and *BIO_ctrl_get_read_request()* return the amount of data requested, or the buffer size if it is less, if the last read attempt at the other half of the BIO pair failed due to an empty buffer. This can be used to determine how much data should be written to the BIO so the next read will succeed: this is most useful in TLS/SSL applications where the amount of data read is usually meaningful rather than just a buffer size. After a successful read this call will return zero. It also will return zero once new data has been written satisfying the read request or part of it. Note that *BIO_get_read_request()* never returns an amount larger than that returned by *BIO_get_write_guarantee()*.

BIO_ctrl_reset_read_request() can also be used to reset the value returned by *BIO_get_read_request()* to zero.

NOTES

Both halves of a BIO pair should be freed. That is even if one half is implicit freed due to a *BIO_free_all()* or *SSL_free()* call the other half needs to be freed.

When used in bidirectional applications (such as TLS/SSL) care should be taken to flush any data in the write buffer. This can be done by calling *BIO_pending()* on the other half of the pair and, if any data is pending, reading it and sending it to the underlying transport. This must be done before any normal processing (such as calling *select()*) due to a request and *BIO_should_read()* being true.

To see why this is important consider a case where a request is sent using *BIO_write()* and a response read with *BIO_read()*, this can occur during an TLS/SSL handshake for example. *BIO_write()* will succeed and place data in the write buffer. *BIO_read()* will initially fail and *BIO_should_read()* will be true. If the application then waits for data to be available on the underlying transport before flushing the write buffer it will never succeed because the request was never sent!

RETURN VALUES

BIO_new_bio_pair() returns 1 on success, with the new BIOs available in **bio1** and **bio2**, or 0 on failure, with NULL pointers stored into the locations for **bio1** and **bio2**. Check the error stack for more information.

[XXXXX: More return values need to be added here]

EXAMPLE

The BIO pair can be used to have full control over the network access of an application. The application can call *select()* on the socket as required without having to go through the SSL−interface.

```
BIO *internal_bio, *network_bio;
...
BIO_new_bio_pair(internal_bio, 0, network_bio, 0);
SSL_set_bio(ssl, internal_bio, internal_bio);
SSL_operations();
...
application |   TLS-engine
   |        |
   +----------> SSL_operations()
            |     /\    ||
            |     ||    \/
            |   BIO-pair (internal_bio)
   +----------< BIO-pair (network_bio)
   |        |
 socket     |
```

```
...
SSL_free(ssl);                  /* implicitly frees internal_bio */
BIO_free(network_bio);
...
```

As the BIO pair will only buffer the data and never directly access the connection, it behaves non-blocking and will return as soon as the write buffer is full or the read buffer is drained. Then the application has to flush the write buffer and/or fill the read buffer.

Use the *BIO_ctrl_pending()*, to find out whether data is buffered in the BIO and must be transfered to the network. Use *BIO_ctrl_get_read_request()* to find out, how many bytes must be written into the buffer before the *SSL_operation()* can successfully be continued.

WARNING

As the data is buffered, *SSL_operation()* may return with a ERROR_SSL_WANT_READ condition, but there is still data in the write buffer. An application must not rely on the error value of *SSL_operation()* but must assure that the write buffer is always flushed first. Otherwise a deadlock may occur as the peer might be waiting for the data before being able to continue.

SEE ALSO

SSL_set_bio (3), *ssl* (3), *openssl_bio* (3), *BIO_should_retry* (3), *BIO_read* (3)

NAME

BIO_s_connect, BIO_set_conn_hostname, BIO_set_conn_port, BIO_set_conn_ip, BIO_set_conn_int_port, BIO_get_conn_hostname, BIO_get_conn_port, BIO_get_conn_ip, BIO_get_conn_int_port, BIO_set_nbio, BIO_do_connect − connect BIO

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD * BIO_s_connect(void);

BIO *BIO_new_connect(char *name);

long BIO_set_conn_hostname(BIO *b, char *name);
long BIO_set_conn_port(BIO *b, char *port);
long BIO_set_conn_ip(BIO *b, char *ip);
long BIO_set_conn_int_port(BIO *b, char *port);
char *BIO_get_conn_hostname(BIO *b);
char *BIO_get_conn_port(BIO *b);
char *BIO_get_conn_ip(BIO *b, dummy);
long BIO_get_conn_int_port(BIO *b, int port);

long BIO_set_nbio(BIO *b, long n);

int BIO_do_connect(BIO *b);
```

DESCRIPTION

BIO_s_connect() returns the connect BIO method. This is a wrapper round the platform's TCP/IP socket connection routines.

Using connect BIOs, TCP/IP connections can be made and data transferred using only BIO routines. In this way any platform specific operations are hidden by the BIO abstraction.

Read and write operations on a connect BIO will perform I/O on the underlying connection. If no connection is established and the port and hostname (see below) is set up properly then a connection is established first.

Connect BIOs support *BIO_puts()* but not *BIO_gets()*.

If the close flag is set on a connect BIO then any active connection is shutdown and the socket closed when the BIO is freed.

Calling *BIO_reset()* on a connect BIO will close any active connection and reset the BIO into a state where it can connect to the same host again.

BIO_get_fd() places the underlying socket in **c** if it is not NULL, it also returns the socket . If **c** is not NULL it should be of type (int *).

BIO_set_conn_hostname() uses the string **name** to set the hostname. The hostname can be an IP address. The hostname can also include the port in the form hostname:port . It is also acceptable to use the form ''hostname/any/other/path'' or ''hostname:port/any/other/path''.

BIO_set_conn_port() sets the port to **port**. **port** can be the numerical form or a string such as ''http''. A string will be looked up first using *getservbyname()* on the host platform but if that fails a standard table of port names will be used. Currently the list is http, telnet, socks, https, ssl, ftp, gopher and wais.

BIO_set_conn_ip() sets the IP address to **ip** using binary form, that is four bytes specifying the IP address in big-endian form.

BIO_set_conn_int_port() sets the port using **port**. **port** should be of type (int *).

BIO_get_conn_hostname() returns the hostname of the connect BIO or NULL if the BIO is initialized but no hostname is set. This return value is an internal pointer which should not be modified.

BIO_get_conn_port() returns the port as a string.

BIO_get_conn_ip() returns the IP address in binary form.

BIO_get_conn_int_port() returns the port as an int.

BIO_set_nbio() sets the non blocking I/O flag to **n**. If **n** is zero then blocking I/O is set. If **n** is 1 then non blocking I/O is set. Blocking I/O is the default. The call to *BIO_set_nbio()* should be made before the connection is established because non blocking I/O is set during the connect process.

BIO_new_connect() combines *BIO_new()* and *BIO_set_conn_hostname()* into a single call: that is it creates a new connect BIO with **name**.

BIO_do_connect() attempts to connect the supplied BIO. It returns 1 if the connection was established successfully. A zero or negative value is returned if the connection could not be established, the call *BIO_should_retry()* should be used for non blocking connect BIOs to determine if the call should be retried.

NOTES

If blocking I/O is set then a non positive return value from any I/O call is caused by an error condition, although a zero return will normally mean that the connection was closed.

If the port name is supplied as part of the host name then this will override any value set with *BIO_set_conn_port()*. This may be undesirable if the application does not wish to allow connection to arbitrary ports. This can be avoided by checking for the presence of the ':' character in the passed hostname and either indicating an error or truncating the string at that point.

The values returned by *BIO_get_conn_hostname()*, *BIO_get_conn_port()*, *BIO_get_conn_ip()* and *BIO_get_conn_int_port()* are updated when a connection attempt is made. Before any connection attempt the values returned are those set by the application itself.

Applications do not have to call *BIO_do_connect()* but may wish to do so to separate the connection process from other I/O processing.

If non blocking I/O is set then retries will be requested as appropriate.

It addition to *BIO_should_read()* and *BIO_should_write()* it is also possible for *BIO_should_io_special()* to be true during the initial connection process with the reason BIO_RR_CONNECT. If this is returned then this is an indication that a connection attempt would block, the application should then take appropriate action to wait until the underlying socket has connected and retry the call.

BIO_set_conn_hostname(), *BIO_set_conn_port()*, *BIO_set_conn_ip()*, *BIO_set_conn_int_port()*, *BIO_get_conn_hostname()*, *BIO_get_conn_port()*, *BIO_get_conn_ip()*, *BIO_get_conn_int_port()*, *BIO_set_nbio()* and *BIO_do_connect()* are macros.

RETURN VALUES

BIO_s_connect() returns the connect BIO method.

BIO_get_fd() returns the socket or −1 if the BIO has not been initialized.

BIO_set_conn_hostname(), *BIO_set_conn_port()*, *BIO_set_conn_ip()* and *BIO_set_conn_int_port()* always return 1.

BIO_get_conn_hostname() returns the connected hostname or NULL is none was set.

BIO_get_conn_port() returns a string representing the connected port or NULL if not set.

BIO_get_conn_ip() returns a pointer to the connected IP address in binary form or all zeros if not set.

BIO_get_conn_int_port() returns the connected port or 0 if none was set.

BIO_set_nbio() always returns 1.

BIO_do_connect() returns 1 if the connection was successfully established and 0 or −1 if the connection failed.

EXAMPLE

This is example connects to a webserver on the local host and attempts to retrieve a page and copy the result to standard output.

```
BIO *cbio, *out;
int len;
char tmpbuf[1024];
ERR_load_crypto_strings();
cbio = BIO_new_connect("localhost:http");
out = BIO_new_fp(stdout, BIO_NOCLOSE);
if(BIO_do_connect(cbio) <= 0) {
        fprintf(stderr, "Error connecting to server\n");
        ERR_print_errors_fp(stderr);
        /* whatever ... */
        }
BIO_puts(cbio, "GET / HTTP/1.0\n\n");
for(;;) {
        len = BIO_read(cbio, tmpbuf, 1024);
        if(len <= 0) break;
        BIO_write(out, tmpbuf, len);
}
BIO_free(cbio);
BIO_free(out);
```

SEE ALSO

TBA

NAME

BIO_s_fd, BIO_set_fd, BIO_get_fd, BIO_new_fd − file descriptor BIO

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *   BIO_s_fd(void);

#define BIO_set_fd(b,fd,c)     BIO_int_ctrl(b,BIO_C_SET_FD,c,fd)
#define BIO_get_fd(b,c)        BIO_ctrl(b,BIO_C_GET_FD,0,(char *)c)

BIO *BIO_new_fd(int fd, int close_flag);
```

DESCRIPTION

BIO_s_fd() returns the file descriptor BIO method. This is a wrapper round the platforms file descriptor routines such as *read()* and *write()*.

BIO_read() and *BIO_write()* read or write the underlying descriptor. *BIO_puts()* is supported but *BIO_gets()* is not.

If the close flag is set then then *close()* is called on the underlying file descriptor when the BIO is freed.

BIO_reset() attempts to change the file pointer to the start of file using lseek(fd, 0, 0).

BIO_seek() sets the file pointer to position **ofs** from start of file using lseek(fd, ofs, 0).

BIO_tell() returns the current file position by calling lseek(fd, 0, 1).

BIO_set_fd() sets the file descriptor of BIO **b** to **fd** and the close flag to **c**.

BIO_get_fd() places the file descriptor in **c** if it is not NULL, it also returns the file descriptor. If **c** is not NULL it should be of type (int *).

BIO_new_fd() returns a file descriptor BIO using **fd** and **close_flag**.

NOTES

The behaviour of *BIO_read()* and *BIO_write()* depends on the behavior of the platforms *read()* and *write()* calls on the descriptor. If the underlying file descriptor is in a non blocking mode then the BIO will behave in the manner described in the *BIO_read* (3) and *BIO_should_retry* (3) manual pages.

File descriptor BIOs should not be used for socket I/O. Use socket BIOs instead.

RETURN VALUES

BIO_s_fd() returns the file descriptor BIO method.

BIO_reset() returns zero for success and −1 if an error occurred. *BIO_seek()* and *BIO_tell()* return the current file position or −1 is an error occurred. These values reflect the underlying *lseek()* behaviour.

BIO_set_fd() always returns 1.

BIO_get_fd() returns the file descriptor or −1 if the BIO has not been initialized.

BIO_new_fd() returns the newly allocated BIO or NULL is an error occurred.

EXAMPLE

This is a file descriptor BIO version of ‘‘Hello World’’:

```
BIO *out;
out = BIO_new_fd(fileno(stdout), BIO_NOCLOSE);
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

SEE ALSO

BIO_seek (3), *BIO_tell* (3), *BIO_reset* (3), *BIO_read* (3), *BIO_write* (3), *BIO_puts* (3), *BIO_gets* (3), *BIO_printf* (3), *BIO_set_close* (3), *BIO_get_close* (3)

NAME

BIO_s_file, BIO_new_file, BIO_new_fp, BIO_set_fp, BIO_get_fp, BIO_read_filename, BIO_write_filename, BIO_append_filename, BIO_rw_filename − FILE bio

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *   BIO_s_file(void);
BIO *BIO_new_file(const char *filename, const char *mode);
BIO *BIO_new_fp(FILE *stream, int flags);

BIO_set_fp(BIO *b,FILE *fp, int flags);
BIO_get_fp(BIO *b,FILE **fpp);

int BIO_read_filename(BIO *b, char *name)
int BIO_write_filename(BIO *b, char *name)
int BIO_append_filename(BIO *b, char *name)
int BIO_rw_filename(BIO *b, char *name)
```

DESCRIPTION

BIO_s_file() returns the BIO file method. As its name implies it is a wrapper round the stdio FILE structure and it is a source/sink BIO.

Calls to *BIO_read()* and *BIO_write()* read and write data to the underlying stream. *BIO_gets()* and *BIO_puts()* are supported on file BIOs.

BIO_flush() on a file BIO calls the *fflush()* function on the wrapped stream.

BIO_reset() attempts to change the file pointer to the start of file using fseek(stream, 0, 0).

BIO_seek() sets the file pointer to position **ofs** from start of file using fseek(stream, ofs, 0).

BIO_eof() calls *feof()*.

Setting the BIO_CLOSE flag calls *fclose()* on the stream when the BIO is freed.

BIO_new_file() creates a new file BIO with mode **mode** the meaning of **mode** is the same as the stdio function *fopen()*. The BIO_CLOSE flag is set on the returned BIO.

BIO_new_fp() creates a file BIO wrapping **stream**. Flags can be: BIO_CLOSE, BIO_NOCLOSE (the close flag) BIO_FP_TEXT (sets the underlying stream to text mode, default is binary: this only has any effect under Win32).

BIO_set_fp() set the fp of a file BIO to **fp**. **flags** has the same meaning as in *BIO_new_fp()*, it is a macro.

BIO_get_fp() retrieves the fp of a file BIO, it is a macro.

BIO_seek() is a macro that sets the position pointer to **offset** bytes from the start of file.

BIO_tell() returns the value of the position pointer.

BIO_read_filename(), *BIO_write_filename()*, *BIO_append_filename()* and *BIO_rw_filename()* set the file BIO **b** to use file **name** for reading, writing, append or read write respectively.

NOTES

When wrapping stdout, stdin or stderr the underlying stream should not normally be closed so the BIO_NOCLOSE flag should be set.

Because the file BIO calls the underlying stdio functions any quirks in stdio behaviour will be mirrored by the corresponding BIO.

EXAMPLES

File BIO ‘‘hello world’’:

```
BIO *bio_out;
bio_out = BIO_new_fp(stdout, BIO_NOCLOSE);
BIO_printf(bio_out, "Hello World\n");
```

Alternative technique:

```
BIO *bio_out;
bio_out = BIO_new(BIO_s_file());
if(bio_out == NULL) /* Error ... */
if(!BIO_set_fp(bio_out, stdout, BIO_NOCLOSE)) /* Error ... */
BIO_printf(bio_out, "Hello World\n");
```

Write to a file:

```
BIO *out;
out = BIO_new_file("filename.txt", "w");
if(!out) /* Error occurred */
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

Alternative technique:

```
BIO *out;
out = BIO_new(BIO_s_file());
if(out == NULL) /* Error ... */
if(!BIO_write_filename(out, "filename.txt")) /* Error ... */
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

RETURN VALUES

BIO_s_file() returns the file BIO method.

BIO_new_file() and *BIO_new_fp()* return a file BIO or NULL if an error occurred.

BIO_set_fp() and *BIO_get_fp()* return 1 for success or 0 for failure (although the current implementation never return 0).

BIO_seek() returns the same value as the underlying *fseek()* function: 0 for success or −1 for failure.

BIO_tell() returns the current file position.

BIO_read_filename(), *BIO_write_filename()*, *BIO_append_filename()* and *BIO_rw_filename()* return 1 for success or 0 for failure.

BUGS

BIO_reset() and *BIO_seek()* are implemented using *fseek()* on the underlying stream. The return value for *fseek()* is 0 for success or −1 if an error occurred this differs from other types of BIO which will typically return 1 for success and a non positive value if an error occurred.

SEE ALSO

BIO_seek (3), *BIO_tell* (3), *BIO_reset* (3), *BIO_flush* (3), *BIO_read* (3), *BIO_write* (3), *BIO_puts* (3), *BIO_gets* (3), *BIO_printf* (3), *BIO_set_close* (3), *BIO_get_close* (3)

NAME

BIO_s_mem, BIO_set_mem_eof_return, BIO_get_mem_data, BIO_set_mem_buf, BIO_get_mem_ptr, BIO_new_mem_buf − memory BIO

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *   BIO_s_mem(void);

BIO_set_mem_eof_return(BIO *b,int v)
long BIO_get_mem_data(BIO *b, char **pp)
BIO_set_mem_buf(BIO *b,BUF_MEM *bm,int c)
BIO_get_mem_ptr(BIO *b,BUF_MEM **pp)

BIO *BIO_new_mem_buf(void *buf, int len);
```

DESCRIPTION

BIO_s_mem() return the memory BIO method function.

A memory BIO is a source/sink BIO which uses memory for its I/O. Data written to a memory BIO is stored in a BUF_MEM structure which is extended as appropriate to accommodate the stored data.

Any data written to a memory BIO can be recalled by reading from it. Unless the memory BIO is read only any data read from it is deleted from the BIO.

Memory BIOs support *BIO_gets()* and *BIO_puts()*.

If the BIO_CLOSE flag is set when a memory BIO is freed then the underlying BUF_MEM structure is also freed.

Calling *BIO_reset()* on a read write memory BIO clears any data in it. On a read only BIO it restores the BIO to its original state and the read only data can be read again.

BIO_eof() is true if no data is in the BIO.

BIO_ctrl_pending() returns the number of bytes currently stored.

BIO_set_mem_eof_return() sets the behaviour of memory BIO **b** when it is empty. If the **v** is zero then an empty memory BIO will return EOF (that is it will return zero and BIO_should_retry(b) will be false. If **v** is non zero then it will return **v** when it is empty and it will set the read retry flag (that is BIO_read_retry(b) is true). To avoid ambiguity with a normal positive return value **v** should be set to a negative value, typically −1.

BIO_get_mem_data() sets **pp** to a pointer to the start of the memory BIOs data and returns the total amount of data available. It is implemented as a macro.

BIO_set_mem_buf() sets the internal BUF_MEM structure to **bm** and sets the close flag to **c**, that is **c** should be either BIO_CLOSE or BIO_NOCLOSE. It is a macro.

BIO_get_mem_ptr() places the underlying BUF_MEM structure in **pp**. It is a macro.

BIO_new_mem_buf() creates a memory BIO using **len** bytes of data at **buf**, if **len** is −1 then the **buf** is assumed to be null terminated and its length is determined by **strlen**. The BIO is set to a read only state and as a result cannot be written to. This is useful when some data needs to be made available from a static area of memory in the form of a BIO. The supplied data is read directly from the supplied buffer: it is **not** copied first, so the supplied area of memory must be unchanged until the BIO is freed.

NOTES

Writes to memory BIOs will always succeed if memory is available: that is their size can grow indefinitely.

Every read from a read write memory BIO will remove the data just read with an internal copy operation, if a BIO contains a lots of data and it is read in small chunks the operation can be very slow. The use of a read only memory BIO avoids this problem. If the BIO must be read write then adding a buffering BIO to the

chain will speed up the process.

BUGS

There should be an option to set the maximum size of a memory BIO.

There should be a way to ‘‘rewind’’ a read write BIO without destroying its contents.

The copying operation should not occur after every small read of a large BIO to improve efficiency.

EXAMPLE

Create a memory BIO and write some data to it:

```
BIO *mem = BIO_new(BIO_s_mem());
BIO_puts(mem, "Hello World\n");
```

Create a read only memory BIO:

```
char data[] = "Hello World";
BIO *mem;
mem = BIO_new_mem_buf(data, -1);
```

Extract the BUF_MEM structure from a memory BIO and then free up the BIO:

```
BUF_MEM *bptr;
BIO_get_mem_ptr(mem, &bptr);
BIO_set_close(mem, BIO_NOCLOSE); /* So BIO_free() leaves BUF_MEM alone */
BIO_free(mem);
```

SEE ALSO

TBA

NAME

BIO_s_null − null data sink

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *   BIO_s_null(void);
```

DESCRIPTION

BIO_s_null() returns the null sink BIO method. Data written to the null sink is discarded, reads return EOF.

NOTES

A null sink BIO behaves in a similar manner to the Unix /dev/null device.

A null bio can be placed on the end of a chain to discard any data passed through it.

A null sink is useful if, for example, an application wishes to digest some data by writing through a digest bio but not send the digested data anywhere. Since a BIO chain must normally include a source/sink BIO this can be achieved by adding a null sink BIO to the end of the chain

RETURN VALUES

BIO_s_null() returns the null sink BIO method.

SEE ALSO

TBA

NAME

BIO_s_socket, BIO_new_socket − socket BIO

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *BIO_s_socket(void);

long BIO_set_fd(BIO *b, int fd, long close_flag);
long BIO_get_fd(BIO *b, int *c);

BIO *BIO_new_socket(int sock, int close_flag);
```

DESCRIPTION

BIO_s_socket() returns the socket BIO method. This is a wrapper round the platform's socket routines.

BIO_read() and *BIO_write()* read or write the underlying socket. *BIO_puts()* is supported but *BIO_gets()* is not.

If the close flag is set then the socket is shut down and closed when the BIO is freed.

BIO_set_fd() sets the socket of BIO **b** to **fd** and the close flag to **close_flag**.

BIO_get_fd() places the socket in **c** if it is not NULL, it also returns the socket. If **c** is not NULL it should be of type (int *).

BIO_new_socket() returns a socket BIO using **sock** and **close_flag**.

NOTES

Socket BIOs also support any relevant functionality of file descriptor BIOs.

The reason for having separate file descriptor and socket BIOs is that on some platforms sockets are not file descriptors and use distinct I/O routines, Windows is one such platform. Any code mixing the two will not work on all platforms.

BIO_set_fd() and *BIO_get_fd()* are macros.

RETURN VALUES

BIO_s_socket() returns the socket BIO method.

BIO_set_fd() always returns 1.

BIO_get_fd() returns the socket or −1 if the BIO has not been initialized.

BIO_new_socket() returns the newly allocated BIO or NULL is an error occurred.

SEE ALSO

TBA

NAME

BIO_set_callback, BIO_get_callback, BIO_set_callback_arg, BIO_get_callback_arg, BIO_debug_callback − BIO callback functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

#define BIO_set_callback(b,cb)          ((b)->callback=(cb))
#define BIO_get_callback(b)             ((b)->callback)
#define BIO_set_callback_arg(b,arg)     ((b)->cb_arg=(char *)(arg))
#define BIO_get_callback_arg(b)                 ((b)->cb_arg)

long BIO_debug_callback(BIO *bio,int cmd,const char *argp,int argi,
       long argl,long ret);

typedef long (*callback)(BIO *b, int oper, const char *argp,
                         int argi, long argl, long retvalue);
```

DESCRIPTION

BIO_set_callback() and *BIO_get_callback()* set and retrieve the BIO callback, they are both macros. The callback is called during most high level BIO operations. It can be used for debugging purposes to trace operations on a BIO or to modify its operation.

BIO_set_callback_arg() and *BIO_get_callback_arg()* are macros which can be used to set and retrieve an argument for use in the callback.

BIO_debug_callback() is a standard debugging callback which prints out information relating to each BIO operation. If the callback argument is set if is interpreted as a BIO to send the information to, otherwise stderr is used.

callback() is the callback function itself. The meaning of each argument is described below.

The BIO the callback is attached to is passed in **b**.

oper is set to the operation being performed. For some operations the callback is called twice, once before and once after the actual operation, the latter case has **oper** or'ed with BIO_CB_RETURN.

The meaning of the arguments **argp**, **argi** and **argl** depends on the value of **oper**, that is the operation being performed.

retvalue is the return value that would be returned to the application if no callback were present. The actual value returned is the return value of the callback itself. In the case of callbacks called before the actual BIO operation 1 is placed in retvalue, if the return value is not positive it will be immediately returned to the application and the BIO operation will not be performed.

The callback should normally simply return **retvalue** when it has finished processing, unless if specifically wishes to modify the value returned to the application.

CALLBACK OPERATIONS

BIO_free(b)

callback(b, BIO_CB_FREE, NULL, 0L, 0L, 1L) is called before the free operation.

BIO_read(b, out, outl)

callback(b, BIO_CB_READ, out, outl, 0L, 1L) is called before the read and callback(b, BIO_CB_READ|BIO_CB_RETURN, out, outl, 0L, retvalue) after.

BIO_write(b, in, inl)

callback(b, BIO_CB_WRITE, in, inl, 0L, 1L) is called before the write and callback(b, BIO_CB_WRITE|BIO_CB_RETURN, in, inl, 0L, retvalue) after.

BIO_gets(b, out, outl)
callback(b, BIO_CB_GETS, out, outl, 0L, 1L) is called before the operation and callback(b, BIO_CB_GETS|BIO_CB_RETURN, out, outl, 0L, retvalue) after.

BIO_puts(b, in)
callback(b, BIO_CB_WRITE, in, 0, 0L, 1L) is called before the operation and callback(b, BIO_CB_WRITE|BIO_CB_RETURN, in, 0, 0L, retvalue) after.

BIO_ctrl(BIO *b, int cmd, long larg, void *parg)
callback(b,BIO_CB_CTRL,parg,cmd,larg,1L) is called before the call and callback(b,BIO_CB_CTRL|BIO_CB_RETURN,parg,cmd, larg,ret) after.

EXAMPLE

The *BIO_debug_callback()* function is a good example, its source is in crypto/bio/bio_cb.c

SEE ALSO

TBA

NAME

BIO_should_retry, BIO_should_read, BIO_should_write, BIO_should_io_special, BIO_retry_type, BIO_should_retry, BIO_get_retry_BIO, BIO_get_retry_reason − BIO retry functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>

#define BIO_should_read(a)              ((a)->flags & BIO_FLAGS_READ)
#define BIO_should_write(a)             ((a)->flags & BIO_FLAGS_WRITE)
#define BIO_should_io_special(a)        ((a)->flags & BIO_FLAGS_IO_SPECIAL)
#define BIO_retry_type(a)               ((a)->flags & BIO_FLAGS_RWS)
#define BIO_should_retry(a)             ((a)->flags & BIO_FLAGS_SHOULD_RETRY)

#define BIO_FLAGS_READ          0x01
#define BIO_FLAGS_WRITE         0x02
#define BIO_FLAGS_IO_SPECIAL    0x04
#define BIO_FLAGS_RWS (BIO_FLAGS_READ|BIO_FLAGS_WRITE|BIO_FLAGS_IO_SPECIAL)
#define BIO_FLAGS_SHOULD_RETRY 0x08

BIO *  BIO_get_retry_BIO(BIO *bio, int *reason);
int    BIO_get_retry_reason(BIO *bio);
```

DESCRIPTION

These functions determine why a BIO is not able to read or write data. They will typically be called after a failed *BIO_read()* or *BIO_write()* call.

BIO_should_retry() is true if the call that produced this condition should then be retried at a later time.

If *BIO_should_retry()* is false then the cause is an error condition.

BIO_should_read() is true if the cause of the condition is that a BIO needs to read data.

BIO_should_write() is true if the cause of the condition is that a BIO needs to read data.

BIO_should_io_special() is true if some ‘‘special’’ condition, that is a reason other than reading or writing is the cause of the condition.

BIO_get_retry_reason() returns a mask of the cause of a retry condition consisting of the values **BIO_FLAGS_READ**, **BIO_FLAGS_WRITE**, **BIO_FLAGS_IO_SPECIAL** though current BIO types will only set one of these.

BIO_get_retry_BIO() determines the precise reason for the special condition, it returns the BIO that caused this condition and if **reason** is not NULL it contains the reason code. The meaning of the reason code and the action that should be taken depends on the type of BIO that resulted in this condition.

BIO_get_retry_reason() returns the reason for a special condition if passed the relevant BIO, for example as returned by *BIO_get_retry_BIO()*.

NOTES

If *BIO_should_retry()* returns false then the precise ‘‘error condition’’ depends on the BIO type that caused it and the return code of the BIO operation. For example if a call to *BIO_read()* on a socket BIO returns 0 and *BIO_should_retry()* is false then the cause will be that the connection closed. A similar condition on a file BIO will mean that it has reached EOF. Some BIO types may place additional information on the error queue. For more details see the individual BIO type manual pages.

If the underlying I/O structure is in a blocking mode almost all current BIO types will not request a retry, because the underlying I/O calls will not. If the application knows that the BIO type will never signal a retry then it need not call *BIO_should_retry()* after a failed BIO I/O call. This is typically done with file BIOs.

SSL BIOs are the only current exception to this rule: they can request a retry even if the underlying I/O structure is blocking, if a handshake occurs during a call to *BIO_read()*. An application can retry the failed call immediately or avoid this situation by setting SSL_MODE_AUTO_RETRY on the underlying SSL

structure.

While an application may retry a failed non blocking call immediately this is likely to be very inefficient because the call will fail repeatedly until data can be processed or is available. An application will normally wait until the necessary condition is satisfied. How this is done depends on the underlying I/O structure.

For example if the cause is ultimately a socket and *BIO_should_read()* is true then a call to *select()* may be made to wait until data is available and then retry the BIO operation. By combining the retry conditions of several non blocking BIOs in a single *select()* call it is possible to service several BIOs in a single thread, though the performance may be poor if SSL BIOs are present because long delays can occur during the initial handshake process.

It is possible for a BIO to block indefinitely if the underlying I/O structure cannot process or return any data. This depends on the behaviour of the platforms I/O functions. This is often not desirable: one solution is to use non blocking I/O and use a timeout on the *select()* (or equivalent) call.

BUGS

The OpenSSL ASN1 functions cannot gracefully deal with non blocking I/O: that is they cannot retry after a partial read or write. This is usually worked around by only passing the relevant data to ASN1 functions when the entire structure can be read or written.

SEE ALSO

TBA

NAME

BN_BLINDING_new, BN_BLINDING_free, BN_BLINDING_update, BN_BLINDING_convert, BN_BLINDING_invert, BN_BLINDING_convert_ex, BN_BLINDING_invert_ex, BN_BLINDING_set_thread, BN_BLINDING_cmp_thread, BN_BLINDING_get_flags, BN_BLINDING_set_flags, BN_BLINDING_create_param − blinding related BIGNUM functions.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

BN_BLINDING *BN_BLINDING_new(const BIGNUM *A, const BIGNUM *Ai,
       BIGNUM *mod);
void BN_BLINDING_free(BN_BLINDING *b);
int BN_BLINDING_update(BN_BLINDING *b,BN_CTX *ctx);
int BN_BLINDING_convert(BIGNUM *n, BN_BLINDING *b, BN_CTX *ctx);
int BN_BLINDING_invert(BIGNUM *n, BN_BLINDING *b, BN_CTX *ctx);
int BN_BLINDING_convert_ex(BIGNUM *n, BIGNUM *r, BN_BLINDING *b,
       BN_CTX *ctx);
int BN_BLINDING_invert_ex(BIGNUM *n, const BIGNUM *r, BN_BLINDING *b,
       BN_CTX *ctx);

void BN_BLINDING_set_thread(BN_BLINDING *);
int BN_BLINDING_cmp_thread(const BN_BLINDING *,
       const CRYPTO_THREADID *);
unsigned long BN_BLINDING_get_flags(const BN_BLINDING *);
void BN_BLINDING_set_flags(BN_BLINDING *, unsigned long);
BN_BLINDING *BN_BLINDING_create_param(BN_BLINDING *b,
       const BIGNUM *e, BIGNUM *m, BN_CTX *ctx,
       int (*bn_mod_exp)(BIGNUM *r, const BIGNUM *a, const BIGNUM *p,
                         const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *m_ctx),
       BN_MONT_CTX *m_ctx);
```

DESCRIPTION

BN_BLINDING_new() allocates a new **BN_BLINDING** structure and copies the **A** and **Ai** values into the newly created **BN_BLINDING** object.

BN_BLINDING_free() frees the **BN_BLINDING** structure.

BN_BLINDING_update() updates the **BN_BLINDING** parameters by squaring the **A** and **Ai** or, after specific number of uses and if the necessary parameters are set, by re-creating the blinding parameters.

BN_BLINDING_convert_ex() multiplies **n** with the blinding factor **A**. If **r** is not NULL a copy the inverse blinding factor **Ai** will be returned in **r** (this is useful if a **RSA** object is shared amoung several threads). *BN_BLINDING_invert_ex()* multiplies **n** with the inverse blinding factor **Ai**. If **r** is not NULL it will be used as the inverse blinding.

BN_BLINDING_convert() and *BN_BLINDING_invert()* are wrapper functions for *BN_BLINDING_convert_ex()* and *BN_BLINDING_invert_ex()* with **r** set to NULL.

BN_BLINDING_set_thread() and *BN_BLINDING_cmp_thread()* set and compare the ''thread id'' of the **BN_BLINDING** structure, allowing users of the **BN_BLINDING** structure to provide proper locking if needed for multi-threaded use.

BN_BLINDING_get_flags() returns the BN_BLINDING flags. Currently there are two supported flags: **BN_BLINDING_NO_UPDATE** and **BN_BLINDING_NO_RECREATE**. **BN_BLINDING_NO_UPDATE** inhibits the automatic update of the **BN_BLINDING** parameters after each use and **BN_BLINDING_NO_RECREATE** inhibits the automatic re-creation of the **BN_BLINDING** parameters after a fixed number of uses (currently 32). In newly allocated **BN_BLINDING** objects no flags are set. *BN_BLINDING_set_flags()* sets the **BN_BLINDING** parameters flags.

BN_BLINDING_create_param() creates new **BN_BLINDING** parameters using the exponent **e** and the modulus **m**. **bn_mod_exp** and **m_ctx** can be used to pass special functions for exponentiation (normally *BN_mod_exp_mont()* and **BN_MONT_CTX**).

RETURN VALUES

BN_BLINDING_new() returns the newly allocated **BN_BLINDING** structure or NULL in case of an error.

BN_BLINDING_update(), *BN_BLINDING_convert()*, *BN_BLINDING_invert()*, *BN_BLINDING_convert_ex()* and *BN_BLINDING_invert_ex()* return 1 on success and 0 if an error occured.

BN_BLINDING_get_thread_id() returns the thread id (a **unsigned long** value) or 0 if not set. *BN_BLINDING_cmp_thread()* returns 0 if the thread id associated with the **BN_BLINDING** structure equals the provided thread id (which can be obtained by *CRYPTO_THREADID_set()*), otherwise it returns −1 or +1 to indicate the thread ids are different (if the target architecture supports ordering of thread ids, this follows the traditional ''cmp'' semantics of *memcmp()* or *strcmp()*).

BN_BLINDING_get_flags() returns the currently set **BN_BLINDING** flags (a **unsigned long** value).

BN_BLINDING_create_param() returns the newly created **BN_BLINDING** parameters or NULL on error.

SEE ALSO

openssl_bn (3)

HISTORY

BN_BLINDING_convert_ex, BN_BLINDIND_invert_ex, BN_BLINDING_get_thread_id, BN_BLINDING_set_thread_id, BN_BLINDING_set_flags, BN_BLINDING_get_flags and BN_BLINDING_create_param were first introduced in OpenSSL 0.9.8

BN_BLINDING_get_thread_idptr, BN_BLINDING_set_thread_idptr were first introduced in OpenSSL 0.9.9

BN_BLINDING_get_thread_id, BN_BLINDING_set_thread_id, BN_BLINDING_get_thread_idptr, BN_BLINDING_set_thread_idptr were all deprecated in favour of BN_BLINDING_set_thread, BN_BLINDING_cmp_thread which were introduced in OpenSSL 0.9.9

AUTHOR

Nils Larsch for the OpenSSL project (http://www.openssl.org).

NAME

BN_CTX_new, BN_CTX_init, BN_CTX_free − allocate and free BN_CTX structures

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

BN_CTX *BN_CTX_new(void);

void BN_CTX_init(BN_CTX *c);

void BN_CTX_free(BN_CTX *c);
```

DESCRIPTION

A **BN_CTX** is a structure that holds **BIGNUM** temporary variables used by library functions. Since dynamic memory allocation to create **BIGNUM**s is rather expensive when used in conjunction with repeated subroutine calls, the **BN_CTX** structure is used.

BN_CTX_new() allocates and initializes a **BN_CTX** structure. *BN_CTX_init()* initializes an existing uninitialized **BN_CTX**.

BN_CTX_free() frees the components of the **BN_CTX**, and if it was created by *BN_CTX_new()*, also the structure itself. If *BN_CTX_start* (3) has been used on the **BN_CTX**, *BN_CTX_end* (3) must be called before the **BN_CTX** may be freed by *BN_CTX_free()*.

RETURN VALUES

BN_CTX_new() returns a pointer to the **BN_CTX**. If the allocation fails, it returns **NULL** and sets an error code that can be obtained by *ERR_get_error* (3).

BN_CTX_init() and *BN_CTX_free()* have no return values.

SEE ALSO

openssl_bn (3), *ERR_get_error* (3), *BN_add* (3), *BN_CTX_start* (3)

HISTORY

BN_CTX_new() and *BN_CTX_free()* are available in all versions on SSLeay and OpenSSL. *BN_CTX_init()* was added in SSLeay 0.9.1b.

NAME

BN_CTX_start, BN_CTX_get, BN_CTX_end − use temporary BIGNUM variables

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

void BN_CTX_start(BN_CTX *ctx);

BIGNUM *BN_CTX_get(BN_CTX *ctx);

void BN_CTX_end(BN_CTX *ctx);
```

DESCRIPTION

These functions are used to obtain temporary **BIGNUM** variables from a **BN_CTX** (which can been created by using *BN_CTX_new*(3)) in order to save the overhead of repeatedly creating and freeing **BIGNUM**s in functions that are called from inside a loop.

A function must call *BN_CTX_start()* first. Then, *BN_CTX_get()* may be called repeatedly to obtain temporary **BIGNUM**s. All *BN_CTX_get()* calls must be made before calling any other functions that use the **ctx** as an argument.

Finally, *BN_CTX_end()* must be called before returning from the function. When *BN_CTX_end()* is called, the **BIGNUM** pointers obtained from *BN_CTX_get()* become invalid.

RETURN VALUES

BN_CTX_start() and *BN_CTX_end()* return no values.

BN_CTX_get() returns a pointer to the **BIGNUM**, or **NULL** on error. Once *BN_CTX_get()* has failed, the subsequent calls will return **NULL** as well, so it is sufficient to check the return value of the last *BN_CTX_get()* call. In case of an error, an error code is set, which can be obtained by *ERR_get_error*(3).

SEE ALSO

BN_CTX_new(3)

HISTORY

BN_CTX_start(), *BN_CTX_get()* and *BN_CTX_end()* were added in OpenSSL 0.9.5.

NAME

BN_add, BN_sub, BN_mul, BN_sqr, BN_div, BN_mod, BN_nnmod, BN_mod_add, BN_mod_sub, BN_mod_mul, BN_mod_sqr, BN_exp, BN_mod_exp, BN_gcd − arithmetic operations on BIGNUMs

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *b);

int BN_sub(BIGNUM *r, const BIGNUM *a, const BIGNUM *b);

int BN_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);

int BN_sqr(BIGNUM *r, BIGNUM *a, BN_CTX *ctx);

int BN_div(BIGNUM *dv, BIGNUM *rem, const BIGNUM *a, const BIGNUM *d,
        BN_CTX *ctx);

int BN_mod(BIGNUM *rem, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);

int BN_nnmod(BIGNUM *r, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);

int BN_mod_add(BIGNUM *r, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);

int BN_mod_sub(BIGNUM *r, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);

int BN_mod_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);

int BN_mod_sqr(BIGNUM *r, BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);

int BN_exp(BIGNUM *r, BIGNUM *a, BIGNUM *p, BN_CTX *ctx);

int BN_mod_exp(BIGNUM *r, BIGNUM *a, const BIGNUM *p,
        const BIGNUM *m, BN_CTX *ctx);

int BN_gcd(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
```

DESCRIPTION

BN_add() adds *a* and *b* and places the result in *r* (`r=a+b`). *r* may be the same **BIGNUM** as *a* or *b*.

BN_sub() subtracts *b* from *a* and places the result in *r* (`r=a−b`).

BN_mul() multiplies *a* and *b* and places the result in *r* (`r=a*b`). *r* may be the same **BIGNUM** as *a* or *b*. For multiplication by powers of 2, use *BN_lshift* (3).

BN_sqr() takes the square of *a* and places the result in *r* (`r=a^2`). *r* and *a* may be the same **BIGNUM**. This function is faster than BN_mul(r,a,a).

BN_div() divides *a* by *d* and places the result in *dv* and the remainder in *rem* (`dv=a/d, rem=a%d`). Either of *dv* and *rem* may be **NULL**, in which case the respective value is not returned. The result is rounded towards zero; thus if *a* is negative, the remainder will be zero or negative. For division by powers of 2, use *BN_rshift* (3).

BN_mod() corresponds to *BN_div()* with *dv* set to **NULL**.

BN_nnmod() reduces *a* modulo *m* and places the non-negative remainder in *r*.

BN_mod_add() adds *a* to *b* modulo *m* and places the non-negative result in *r*.

BN_mod_sub() subtracts *b* from *a* modulo *m* and places the non-negative result in *r*.

BN_mod_mul() multiplies *a* by *b* and finds the non-negative remainder respective to modulus *m* (`r=(a*b) mod m`). *r* may be the same **BIGNUM** as *a* or *b*. For more efficient algorithms for repeated computations using the same modulus, see *BN_mod_mul_montgomery* (3) and *BN_mod_mul_reciprocal* (3).

BN_mod_sqr() takes the square of *a* modulo **m** and places the result in *r*.

BN_exp() raises *a* to the *p*−th power and places the result in *r* (`r=a^p`). This function is faster than repeated applications of *BN_mul()*.

BN_mod_exp() computes *a* to the *p*−th power modulo *m* (`r=a^p % m`). This function uses less time and space than *BN_exp()*.

BN_gcd() computes the greatest common divisor of *a* and *b* and places the result in *r*. *r* may be the same **BIGNUM** as *a* or *b*.

For all functions, *ctx* is a previously allocated **BN_CTX** used for temporary variables; see *BN_CTX_new* (3).

Unless noted otherwise, the result **BIGNUM** must be different from the arguments.

RETURN VALUES

For all functions, 1 is returned for success, 0 on error. The return value should always be checked (e.g., `if (!BN_add(r,a,b)) goto err;`). The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

openssl_bn (3), *ERR_get_error* (3), *BN_CTX_new* (3), *BN_add_word* (3), *BN_set_bit* (3)

HISTORY

BN_add(), *BN_sub()*, *BN_sqr()*, *BN_div()*, *BN_mod()*, *BN_mod_mul()*, *BN_mod_exp()* and *BN_gcd()* are available in all versions of SSLeay and OpenSSL. The *ctx* argument to *BN_mul()* was added in SSLeay 0.9.1b. *BN_exp()* appeared in SSLeay 0.9.0. *BN_nnmod()*, *BN_mod_add()*, *BN_mod_sub()*, and *BN_mod_sqr()* were added in OpenSSL 0.9.7.

NAME

BN_add_word, BN_sub_word, BN_mul_word, BN_div_word, BN_mod_word − arithmetic functions on BIGNUMs with integers

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

int BN_add_word(BIGNUM *a, BN_ULONG w);

int BN_sub_word(BIGNUM *a, BN_ULONG w);

int BN_mul_word(BIGNUM *a, BN_ULONG w);

BN_ULONG BN_div_word(BIGNUM *a, BN_ULONG w);

BN_ULONG BN_mod_word(const BIGNUM *a, BN_ULONG w);
```

DESCRIPTION

These functions perform arithmetic operations on BIGNUMs with unsigned integers. They are much more efficient than the normal BIGNUM arithmetic operations.

BN_add_word() adds **w** to **a** (`a+=w`).

BN_sub_word() subtracts **w** from **a** (`a−=w`).

BN_mul_word() multiplies **a** and **w** (`a*=w`).

BN_div_word() divides **a** by **w** (`a/=w`) and returns the remainder.

BN_mod_word() returns the remainder of **a** divided by **w** (`a%w`).

For *BN_div_word()* and *BN_mod_word()*, **w** must not be 0.

RETURN VALUES

BN_add_word(), *BN_sub_word()* and *BN_mul_word()* return 1 for success, 0 on error. The error codes can be obtained by *ERR_get_error* (3).

BN_mod_word() and *BN_div_word()* return **a**%**w** on success and **(BN_ULONG)−1** if an error occurred.

SEE ALSO

openssl_bn (3), *ERR_get_error* (3), *BN_add* (3)

HISTORY

BN_add_word() and *BN_mod_word()* are available in all versions of SSLeay and OpenSSL. *BN_div_word()* was added in SSLeay 0.8, and *BN_sub_word()* and *BN_mul_word()* in SSLeay 0.9.0.

Before 0.9.8a the return value for *BN_div_word()* and *BN_mod_word()* in case of an error was 0.

NAME

BN_bn2bin, BN_bin2bn, BN_bn2hex, BN_bn2dec, BN_hex2bn, BN_dec2bn, BN_print, BN_print_fp, BN_bn2mpi, BN_mpi2bn − format conversions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

int BN_bn2bin(const BIGNUM *a, unsigned char *to);
BIGNUM *BN_bin2bn(const unsigned char *s, int len, BIGNUM *ret);

char *BN_bn2hex(const BIGNUM *a);
char *BN_bn2dec(const BIGNUM *a);
int BN_hex2bn(BIGNUM **a, const char *str);
int BN_dec2bn(BIGNUM **a, const char *str);

int BN_print(BIO *fp, const BIGNUM *a);
int BN_print_fp(FILE *fp, const BIGNUM *a);

int BN_bn2mpi(const BIGNUM *a, unsigned char *to);
BIGNUM *BN_mpi2bn(unsigned char *s, int len, BIGNUM *ret);
```

DESCRIPTION

BN_bn2bin() converts the absolute value of **a** into big-endian form and stores it at **to**. **to** must point to BN_num_bytes(**a**) bytes of memory.

BN_bin2bn() converts the positive integer in big-endian form of length **len** at **s** into a **BIGNUM** and places it in **ret**. If **ret** is NULL, a new **BIGNUM** is created.

BN_bn2hex() and *BN_bn2dec()* return printable strings containing the hexadecimal and decimal encoding of **a** respectively. For negative numbers, the string is prefaced with a leading '−'. The string must be freed later using *OPENSSL_free()*.

BN_hex2bn() converts the string **str** containing a hexadecimal number to a **BIGNUM** and stores it in ***bn**. If ***bn** is NULL, a new **BIGNUM** is created. If **bn** is NULL, it only computes the number's length in hexadecimal digits. If the string starts with '−', the number is negative. *BN_dec2bn()* is the same using the decimal system.

BN_print() and *BN_print_fp()* write the hexadecimal encoding of **a**, with a leading '−' for negative numbers, to the **BIO** or **FILE fp**.

BN_bn2mpi() and *BN_mpi2bn()* convert **BIGNUM**s from and to a format that consists of the number's length in bytes represented as a 4−byte big-endian number, and the number itself in big-endian format, where the most significant bit signals a negative number (the representation of numbers with the MSB set is prefixed with null byte).

BN_bn2mpi() stores the representation of **a** at **to**, where **to** must be large enough to hold the result. The size can be determined by calling BN_bn2mpi(**a**, NULL).

BN_mpi2bn() converts the **len** bytes long representation at **s** to a **BIGNUM** and stores it at **ret**, or in a newly allocated **BIGNUM** if **ret** is NULL.

RETURN VALUES

BN_bn2bin() returns the length of the big-endian number placed at **to**. *BN_bin2bn()* returns the **BIGNUM**, NULL on error.

BN_bn2hex() and *BN_bn2dec()* return a null-terminated string, or NULL on error. *BN_hex2bn()* and *BN_dec2bn()* return the number's length in hexadecimal or decimal digits, and 0 on error.

BN_print_fp() and *BN_print()* return 1 on success, 0 on write errors.

BN_bn2mpi() returns the length of the representation. *BN_mpi2bn()* returns the **BIGNUM**, and NULL on error.

The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

openssl_bn (3), *ERR_get_error* (3), *BN_zero* (3), *ASN1_INTEGER_to_BN* (3), *BN_num_bytes* (3)

HISTORY

BN_bn2bin(), *BN_bin2bn()*, *BN_print_fp()* and *BN_print()* are available in all versions of SSLeay and OpenSSL.

BN_bn2hex(), *BN_bn2dec()*, *BN_hex2bn()*, *BN_dec2bn()*, *BN_bn2mpi()* and *BN_mpi2bn()* were added in SSLeay 0.9.0.

NAME

BN_cmp, BN_ucmp, BN_is_zero, BN_is_one, BN_is_word, BN_is_odd − BIGNUM comparison and test functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

int BN_cmp(BIGNUM *a, BIGNUM *b);
int BN_ucmp(BIGNUM *a, BIGNUM *b);

int BN_is_zero(BIGNUM *a);
int BN_is_one(BIGNUM *a);
int BN_is_word(BIGNUM *a, BN_ULONG w);
int BN_is_odd(BIGNUM *a);
```

DESCRIPTION

BN_cmp() compares the numbers **a** and **b**. *BN_ucmp()* compares their absolute values.

BN_is_zero(), *BN_is_one()* and *BN_is_word()* test if **a** equals 0, 1, or **w** respectively. *BN_is_odd()* tests if a is odd.

BN_is_zero(), *BN_is_one()*, *BN_is_word()* and *BN_is_odd()* are macros.

RETURN VALUES

BN_cmp() returns −1 if **a** < **b**, 0 if **a** == **b** and 1 if **a** > **b**. *BN_ucmp()* is the same using the absolute values of **a** and **b**.

BN_is_zero(), *BN_is_one() BN_is_word()* and *BN_is_odd()* return 1 if the condition is true, 0 otherwise.

SEE ALSO

openssl_bn (3)

HISTORY

BN_cmp(), *BN_ucmp()*, *BN_is_zero()*, *BN_is_one()* and *BN_is_word()* are available in all versions of SSLeay and OpenSSL. *BN_is_odd()* was added in SSLeay 0.8.

NAME

BN_copy, BN_dup − copy BIGNUMs

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

BIGNUM *BN_copy(BIGNUM *to, const BIGNUM *from);

BIGNUM *BN_dup(const BIGNUM *from);
```

DESCRIPTION

BN_copy() copies **from** to **to**. *BN_dup()* creates a new **BIGNUM** containing the value **from**.

RETURN VALUES

BN_copy() returns **to** on success, NULL on error. *BN_dup()* returns the new **BIGNUM**, and NULL on error. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

openssl_bn (3), *ERR_get_error* (3)

HISTORY

BN_copy() and *BN_dup()* are available in all versions of SSLeay and OpenSSL.

NAME

BN_generate_prime, BN_is_prime, BN_is_prime_fasttest − generate primes and test for primality

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

BIGNUM *BN_generate_prime(BIGNUM *ret, int num, int safe, BIGNUM *add,
    BIGNUM *rem, void (*callback)(int, int, void *), void *cb_arg);

int BN_is_prime(const BIGNUM *a, int checks, void (*callback)(int, int,
    void *), BN_CTX *ctx, void *cb_arg);

int BN_is_prime_fasttest(const BIGNUM *a, int checks,
    void (*callback)(int, int, void *), BN_CTX *ctx, void *cb_arg,
    int do_trial_division);
```

DESCRIPTION

BN_generate_prime() generates a pseudo-random prime number of **num** bits. If **ret** is not **NULL**, it will be used to store the number.

If **callback** is not **NULL**, it is called as follows:

- **callback(0, i, cb_arg)** is called after generating the i−th potential prime number.
- While the number is being tested for primality, **callback(1, j, cb_arg)** is called as described below.
- When a prime has been found, **callback(2, i, cb_arg)** is called.

The prime may have to fulfill additional requirements for use in Diffie-Hellman key exchange:

If **add** is not **NULL**, the prime will fulfill the condition p % **add** == **rem** (p % **add** == 1 if **rem** == **NULL**) in order to suit a given generator.

If **safe** is true, it will be a safe prime (i.e. a prime p so that (p−1)/2 is also prime).

The PRNG must be seeded prior to calling *BN_generate_prime()*. The prime number generation has a negligible error probability.

BN_is_prime() and *BN_is_prime_fasttest()* test if the number **a** is prime. The following tests are performed until one of them shows that **a** is composite; if **a** passes all these tests, it is considered prime.

BN_is_prime_fasttest(), when called with **do_trial_division == 1**, first attempts trial division by a number of small primes; if no divisors are found by this test and **callback** is not **NULL**, **callback(1, −1, cb_arg)** is called. If **do_trial_division == 0**, this test is skipped.

Both *BN_is_prime()* and *BN_is_prime_fasttest()* perform a Miller-Rabin probabilistic primality test with **checks** iterations. If **checks == BN_prime_checks**, a number of iterations is used that yields a false positive rate of at most 2^−80 for random input.

If **callback** is not **NULL**, **callback(1, j, cb_arg)** is called after the j−th iteration (j = 0, 1, ...). **ctx** is a preallocated **BN_CTX** (to save the overhead of allocating and freeing the structure in a loop), or **NULL**.

RETURN VALUES

BN_generate_prime() returns the prime number on success, **NULL** otherwise.

BN_is_prime() returns 0 if the number is composite, 1 if it is prime with an error probability of less than 0.25^**checks**, and −1 on error.

The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

openssl_bn (3), *ERR_get_error* (3), *openssl_rand* (3)

HISTORY

The **cb_arg** arguments to *BN_generate_prime()* and to *BN_is_prime()* were added in SSLeay 0.9.0. The **ret** argument to *BN_generate_prime()* was added in SSLeay 0.9.1. *BN_is_prime_fasttest()* was added in OpenSSL 0.9.5.

NAME

BN_mod_inverse − compute inverse modulo n

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

BIGNUM *BN_mod_inverse(BIGNUM *r, BIGNUM *a, const BIGNUM *n,
          BN_CTX *ctx);
```

DESCRIPTION

BN_mod_inverse() computes the inverse of **a** modulo **n** places the result in **r** (`(a*r)%n==1`). If **r** is NULL, a new **BIGNUM** is created.

ctx is a previously allocated **BN_CTX** used for temporary variables. **r** may be the same **BIGNUM** as **a** or **n**.

RETURN VALUES

BN_mod_inverse() returns the **BIGNUM** containing the inverse, and NULL on error. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

openssl_bn (3), *ERR_get_error* (3), *BN_add* (3)

HISTORY

BN_mod_inverse() is available in all versions of SSLeay and OpenSSL.

NAME

BN_mod_mul_montgomery, BN_MONT_CTX_new, BN_MONT_CTX_init, BN_MONT_CTX_free, BN_MONT_CTX_set, BN_MONT_CTX_copy, BN_from_montgomery, BN_to_montgomery − Montgomery multiplication

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

BN_MONT_CTX *BN_MONT_CTX_new(void);
void BN_MONT_CTX_init(BN_MONT_CTX *ctx);
void BN_MONT_CTX_free(BN_MONT_CTX *mont);

int BN_MONT_CTX_set(BN_MONT_CTX *mont, const BIGNUM *m, BN_CTX *ctx);
BN_MONT_CTX *BN_MONT_CTX_copy(BN_MONT_CTX *to, BN_MONT_CTX *from);

int BN_mod_mul_montgomery(BIGNUM *r, BIGNUM *a, BIGNUM *b,
        BN_MONT_CTX *mont, BN_CTX *ctx);

int BN_from_montgomery(BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont,
        BN_CTX *ctx);

int BN_to_montgomery(BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont,
        BN_CTX *ctx);
```

DESCRIPTION

These functions implement Montgomery multiplication. They are used automatically when *BN_mod_exp* (3) is called with suitable input, but they may be useful when several operations are to be performed using the same modulus.

BN_MONT_CTX_new() allocates and initializes a **BN_MONT_CTX** structure. *BN_MONT_CTX_init()* initializes an existing uninitialized **BN_MONT_CTX**.

BN_MONT_CTX_set() sets up the *mont* structure from the modulus *m* by precomputing its inverse and a value R.

BN_MONT_CTX_copy() copies the **BN_MONT_CTX** *from* to *to*.

BN_MONT_CTX_free() frees the components of the **BN_MONT_CTX**, and, if it was created by *BN_MONT_CTX_new()*, also the structure itself.

BN_mod_mul_montgomery() computes Mont(*a*,*b*):=*a**b*Rˆ−1 and places the result in *r*.

BN_from_montgomery() performs the Montgomery reduction *r* = *a**Rˆ−1.

BN_to_montgomery() computes Mont(*a*,Rˆ2), i.e. *a**R. Note that *a* must be non-negative and smaller than the modulus.

For all functions, *ctx* is a previously allocated **BN_CTX** used for temporary variables.

The **BN_MONT_CTX** structure is defined as follows:

```
typedef struct bn_mont_ctx_st
       {
       int ri;         /* number of bits in R */
       BIGNUM RR;      /* Rˆ2 (used to convert to Montgomery form) */
       BIGNUM N;       /* The modulus */
       BIGNUM Ni;      /* R*(1/R mod N) - N*Ni = 1
                        * (Ni is only stored for bignum algorithm) */
       BN_ULONG n0;    /* least significant word of Ni */
       int flags;
       } BN_MONT_CTX;
```

BN_to_montgomery() is a macro.

RETURN VALUES

BN_MONT_CTX_new() returns the newly allocated **BN_MONT_CTX**, and NULL on error.

BN_MONT_CTX_init() and *BN_MONT_CTX_free()* have no return values.

For the other functions, 1 is returned for success, 0 on error. The error codes can be obtained by *ERR_get_error* (3).

WARNING

The inputs must be reduced modulo **m**, otherwise the result will be outside the expected range.

SEE ALSO

openssl_bn (3), *ERR_get_error* (3), *BN_add* (3), *BN_CTX_new* (3)

HISTORY

BN_MONT_CTX_new(), *BN_MONT_CTX_free()*, *BN_MONT_CTX_set()*, *BN_mod_mul_montgomery()*, *BN_from_montgomery()* and *BN_to_montgomery()* are available in all versions of SSLeay and OpenSSL.

BN_MONT_CTX_init() and *BN_MONT_CTX_copy()* were added in SSLeay 0.9.1b.

NAME

BN_mod_mul_reciprocal, BN_div_recp, BN_RECP_CTX_new, BN_RECP_CTX_init, BN_RECP_CTX_free, BN_RECP_CTX_set − modular multiplication using reciprocal

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

BN_RECP_CTX *BN_RECP_CTX_new(void);
void BN_RECP_CTX_init(BN_RECP_CTX *recp);
void BN_RECP_CTX_free(BN_RECP_CTX *recp);

int BN_RECP_CTX_set(BN_RECP_CTX *recp, const BIGNUM *m, BN_CTX *ctx);

int BN_div_recp(BIGNUM *dv, BIGNUM *rem, BIGNUM *a, BN_RECP_CTX *recp,
       BN_CTX *ctx);

int BN_mod_mul_reciprocal(BIGNUM *r, BIGNUM *a, BIGNUM *b,
       BN_RECP_CTX *recp, BN_CTX *ctx);
```

DESCRIPTION

BN_mod_mul_reciprocal() can be used to perform an efficient *BN_mod_mul* (3) operation when the operation will be performed repeatedly with the same modulus. It computes **r**=(**a***__b__)%**m** using **recp**=1/**m**, which is set as described below. **ctx** is a previously allocated **BN_CTX** used for temporary variables.

BN_RECP_CTX_new() allocates and initializes a **BN_RECP** structure. *BN_RECP_CTX_init()* initializes an existing uninitialized **BN_RECP**.

BN_RECP_CTX_free() frees the components of the **BN_RECP**, and, if it was created by *BN_RECP_CTX_new()*, also the structure itself.

BN_RECP_CTX_set() stores **m** in **recp** and sets it up for computing 1/**m** and shifting it left by BN_num_bits(**m**)+1 to make it an integer. The result and the number of bits it was shifted left will later be stored in **recp**.

BN_div_recp() divides **a** by **m** using **recp**. It places the quotient in **dv** and the remainder in **rem**.

The **BN_RECP_CTX** structure is defined as follows:

```
typedef struct bn_recp_ctx_st
       {
       BIGNUM N;       /* the divisor */
       BIGNUM Nr;      /* the reciprocal */
       int num_bits;
       int shift;
       int flags;
       } BN_RECP_CTX;
```

It cannot be shared between threads.

RETURN VALUES

BN_RECP_CTX_new() returns the newly allocated **BN_RECP_CTX**, and NULL on error.

BN_RECP_CTX_init() and *BN_RECP_CTX_free()* have no return values.

For the other functions, 1 is returned for success, 0 on error. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

openssl_bn (3), *ERR_get_error* (3), *BN_add* (3), *BN_CTX_new* (3)

HISTORY

BN_RECP_CTX was added in SSLeay 0.9.0. Before that, the function *BN_reciprocal()* was used instead, and the *BN_mod_mul_reciprocal()* arguments were different.

NAME

BN_new, BN_init, BN_clear, BN_free, BN_clear_free − allocate and free BIGNUMs

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

BIGNUM *BN_new(void);

void BN_init(BIGNUM *);

void BN_clear(BIGNUM *a);

void BN_free(BIGNUM *a);

void BN_clear_free(BIGNUM *a);
```

DESCRIPTION

BN_new() allocates and initializes a **BIGNUM** structure. *BN_init()* initializes an existing uninitialized **BIGNUM**.

BN_clear() is used to destroy sensitive data such as keys when they are no longer needed. It erases the memory used by **a** and sets it to the value 0.

BN_free() frees the components of the **BIGNUM**, and if it was created by *BN_new()*, also the structure itself. *BN_clear_free()* additionally overwrites the data before the memory is returned to the system.

RETURN VALUES

BN_new() returns a pointer to the **BIGNUM**. If the allocation fails, it returns **NULL** and sets an error code that can be obtained by *ERR_get_error* (3).

BN_init(), *BN_clear()*, *BN_free()* and *BN_clear_free()* have no return values.

SEE ALSO

openssl_bn (3), *ERR_get_error* (3)

HISTORY

BN_new(), *BN_clear()*, *BN_free()* and *BN_clear_free()* are available in all versions on SSLeay and OpenSSL. *BN_init()* was added in SSLeay 0.9.1b.

NAME

BN_num_bits, BN_num_bytes, BN_num_bits_word − get BIGNUM size

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

int BN_num_bytes(const BIGNUM *a);

int BN_num_bits(const BIGNUM *a);

int BN_num_bits_word(BN_ULONG w);
```

DESCRIPTION

BN_num_bytes() returns the size of a **BIGNUM** in bytes.

BN_num_bits_word() returns the number of significant bits in a word. If we take 0x00000432 as an example, it returns 11, not 16, not 32. Basically, except for a zero, it returns *floor(log2(w))+1*.

BN_num_bits() returns the number of significant bits in a **BIGNUM**, following the same principle as *BN_num_bits_word()*.

BN_num_bytes() is a macro.

RETURN VALUES

The size.

NOTES

Some have tried using *BN_num_bits()* on individual numbers in RSA keys, DH keys and DSA keys, and found that they don't always come up with the number of bits they expected (something like 512, 1024, 2048, ...). This is because generating a number with some specific number of bits doesn't always set the highest bits, thereby making the number of *significant* bits a little lower. If you want to know the ''key size'' of such a key, either use functions like *RSA_size()*, *DH_size()* and *DSA_size()*, or use *BN_num_bytes()* and multiply with 8 (although there's no real guarantee that will match the ''key size'', just a lot more probability).

SEE ALSO

openssl_bn (3), *DH_size* (3), *DSA_size* (3), *RSA_size* (3)

HISTORY

BN_num_bytes(), *BN_num_bits()* and *BN_num_bits_word()* are available in all versions of SSLeay and OpenSSL.

NAME

BN_rand, BN_pseudo_rand − generate pseudo−random number

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

int BN_rand(BIGNUM *rnd, int bits, int top, int bottom);

int BN_pseudo_rand(BIGNUM *rnd, int bits, int top, int bottom);

int BN_rand_range(BIGNUM *rnd, BIGNUM *range);

int BN_pseudo_rand_range(BIGNUM *rnd, BIGNUM *range);
```

DESCRIPTION

BN_rand() generates a cryptographically strong pseudo-random number of **bits** bits in length and stores it in **rnd**. If **top** is −1, the most significant bit of the random number can be zero. If **top** is 0, it is set to 1, and if **top** is 1, the two most significant bits of the number will be set to 1, so that the product of two such random numbers will always have 2***bits** length. If **bottom** is true, the number will be odd.

BN_pseudo_rand() does the same, but pseudo-random numbers generated by this function are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

BN_rand_range() generates a cryptographically strong pseudo-random number **rnd** in the range 0 <lt>= **rnd** < **range**. *BN_pseudo_rand_range()* does the same, but is based on *BN_pseudo_rand()*, and hence numbers generated by it are not necessarily unpredictable.

The PRNG must be seeded prior to calling *BN_rand()* or *BN_rand_range()*.

RETURN VALUES

The functions return 1 on success, 0 on error. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

openssl_bn (3), *ERR_get_error* (3), *openssl_rand* (3), *RAND_add* (3), *RAND_bytes* (3)

HISTORY

BN_rand() is available in all versions of SSLeay and OpenSSL. *BN_pseudo_rand()* was added in OpenSSL 0.9.5. The **top** == −1 case and the function *BN_rand_range()* were added in OpenSSL 0.9.6a. *BN_pseudo_rand_range()* was added in OpenSSL 0.9.6c.

NAME

BN_set_bit, BN_clear_bit, BN_is_bit_set, BN_mask_bits, BN_lshift, BN_lshift1, BN_rshift, BN_rshift1 − bit operations on BIGNUMs

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

int BN_set_bit(BIGNUM *a, int n);
int BN_clear_bit(BIGNUM *a, int n);

int BN_is_bit_set(const BIGNUM *a, int n);

int BN_mask_bits(BIGNUM *a, int n);

int BN_lshift(BIGNUM *r, const BIGNUM *a, int n);
int BN_lshift1(BIGNUM *r, BIGNUM *a);

int BN_rshift(BIGNUM *r, BIGNUM *a, int n);
int BN_rshift1(BIGNUM *r, BIGNUM *a);
```

DESCRIPTION

BN_set_bit() sets bit **n** in **a** to 1 (`a|=(1<<n)`). The number is expanded if necessary.

BN_clear_bit() sets bit **n** in **a** to 0 (`a&=~(1<<n)`). An error occurs if **a** is shorter than **n** bits.

BN_is_bit_set() tests if bit **n** in **a** is set.

BN_mask_bits() truncates **a** to an **n** bit number (`a&=~((~0)>>n)`). An error occurs if **a** already is shorter than **n** bits.

BN_lshift() shifts **a** left by **n** bits and places the result in **r** (`r=a*2^n`). *BN_lshift1()* shifts **a** left by one and places the result in **r** (`r=2*a`).

BN_rshift() shifts **a** right by **n** bits and places the result in **r** (`r=a/2^n`). *BN_rshift1()* shifts **a** right by one and places the result in **r** (`r=a/2`).

For the shift functions, **r** and **a** may be the same variable.

RETURN VALUES

BN_is_bit_set() returns 1 if the bit is set, 0 otherwise.

All other functions return 1 for success, 0 on error. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

openssl_bn (3), *BN_num_bytes* (3), *BN_add* (3)

HISTORY

BN_set_bit(), *BN_clear_bit()*, *BN_is_bit_set()*, *BN_mask_bits()*, *BN_lshift()*, *BN_lshift1()*, *BN_rshift()*, and *BN_rshift1()* are available in all versions of SSLeay and OpenSSL.

NAME

BN_swap − exchange BIGNUMs

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

void BN_swap(BIGNUM *a, BIGNUM *b);
```

DESCRIPTION

BN_swap() exchanges the values of *a* and *b*.

openssl_bn (3)

HISTORY

BN_swap was added in OpenSSL 0.9.7.

NAME

BN_zero, BN_one, BN_value_one, BN_set_word, BN_get_word − BIGNUM assignment operations

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

int BN_zero(BIGNUM *a);
int BN_one(BIGNUM *a);

const BIGNUM *BN_value_one(void);

int BN_set_word(BIGNUM *a, unsigned long w);
unsigned long BN_get_word(BIGNUM *a);
```

DESCRIPTION

BN_zero(), *BN_one()* and *BN_set_word()* set **a** to the values 0, 1 and **w** respectively. *BN_zero()* and *BN_one()* are macros.

BN_value_one() returns a **BIGNUM** constant of value 1. This constant is useful for use in comparisons and assignment.

BN_get_word() returns **a**, if it can be represented as an unsigned long.

RETURN VALUES

BN_get_word() returns the value **a**, and 0xffffffffL if **a** cannot be represented as an unsigned long.

BN_zero(), *BN_one()* and *BN_set_word()* return 1 on success, 0 otherwise. *BN_value_one()* returns the constant.

BUGS

Someone might change the constant.

If a **BIGNUM** is equal to 0xffffffffL it can be represented as an unsigned long but this value is also returned on error.

SEE ALSO

openssl_bn (3), *BN_bn2bin* (3)

HISTORY

BN_zero(), *BN_one()* and *BN_set_word()* are available in all versions of SSLeay and OpenSSL. *BN_value_one()* and *BN_get_word()* were added in SSLeay 0.8.

BN_value_one() was changed to return a true const BIGNUM * in OpenSSL 0.9.7.

NAME

CONF_modules_free, CONF_modules_finish, CONF_modules_unload - OpenSSL configuration cleanup functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/conf.h>

void CONF_modules_free(void);
void CONF_modules_finish(void);
void CONF_modules_unload(int all);
```

DESCRIPTION

CONF_modules_free() closes down and frees up all memory allocated by all configuration modules.

CONF_modules_finish() calls each configuration modules **finish** handler to free up any configuration that module may have performed.

CONF_modules_unload() finishes and unloads configuration modules. If **all** is set to **0** only modules loaded from DSOs will be unloads. If **all** is **1** all modules, including builtin modules will be unloaded.

NOTES

Normally applications will only call *CONF_modules_free()* at application to tidy up any configuration performed.

RETURN VALUE

None of the functions return a value.

SEE ALSO

conf (5), *OPENSSL_config* (3), ''*CONF_modules_load_file* (3), *CONF_modules_load_file* (3)''

HISTORY

CONF_modules_free(), CONF_modules_unload(), and *CONF_modules_finish()* first appeared in OpenSSL 0.9.7.

NAME

CONF_modules_load_file, CONF_modules_load - OpenSSL configuration functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/conf.h>

int CONF_modules_load_file(const char *filename, const char *appname,
                           unsigned long flags);
int CONF_modules_load(const CONF *cnf, const char *appname,
                      unsigned long flags);
```

DESCRIPTION

The function *CONF_modules_load_file()* configures OpenSSL using file **filename** and application name **appname**. If **filename** is NULL the standard OpenSSL configuration file is used. If **appname** is NULL the standard OpenSSL application name **openssl_conf** is used. The behaviour can be cutomized using **flags**.

CONF_modules_load() is idential to *CONF_modules_load_file()* except it read configuration information from **cnf**.

NOTES

The following **flags** are currently recognized:

CONF_MFLAGS_IGNORE_ERRORS if set errors returned by individual configuration modules are ignored. If not set the first module error is considered fatal and no further modules are loads.

Normally any modules errors will add error information to the error queue. If **CONF_MFLAGS_SILENT** is set no error information is added.

If **CONF_MFLAGS_NO_DSO** is set configuration module loading from DSOs is disabled.

CONF_MFLAGS_IGNORE_MISSING_FILE if set will make *CONF_load_modules_file()* ignore missing configuration files. Normally a missing configuration file return an error.

RETURN VALUE

These functions return 1 for success and a zero or negative value for failure. If module errors are not ignored the return code will reflect the return value of the failing module (this will always be zero or negative).

SEE ALSO

conf (5), *OPENSSL_config* (3), ''*CONF_free* (3), *CONF_free* (3)'', *openssl_err* (3),*openssl_err* (3)

HISTORY

CONF_modules_load_file and CONF_modules_load first appeared in OpenSSL 0.9.7.

NAME

CRYPTO_set_ex_data, CRYPTO_get_ex_data − internal application specific data functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/crypto.h>

int CRYPTO_set_ex_data(CRYPTO_EX_DATA *r, int idx, void *arg);

void *CRYPTO_get_ex_data(CRYPTO_EX_DATA *r, int idx);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

These functions should only be used by applications to manipulate **CRYPTO_EX_DATA** structures passed to the ***new_func()***, ***free_func()*** and ***dup_func()*** callbacks: as passed to ***RSA_get_ex_new_index()*** for example.

CRYPTO_set_ex_data() is used to set application specific data, the data is supplied in the **arg** parameter and its precise meaning is up to the application.

CRYPTO_get_ex_data() is used to retrieve application specific data. The data is returned to the application, this will be the same value as supplied to a previous ***CRYPTO_set_ex_data()*** call.

RETURN VALUES

CRYPTO_set_ex_data() returns 1 on success or 0 on failure.

CRYPTO_get_ex_data() returns the application data or 0 on failure. 0 may also be valid application data but currently it can only fail if given an invalid **idx** parameter.

On failure an error code can be obtained from *ERR_get_error* (3).

SEE ALSO

RSA_get_ex_new_index (3), *DSA_get_ex_new_index* (3), *DH_get_ex_new_index* (3)

HISTORY

CRYPTO_set_ex_data() and *CRYPTO_get_ex_data()* have been available since SSLeay 0.9.0.

NAME

DH_generate_key, DH_compute_key − perform Diffie−Hellman key exchange

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dh.h>

int DH_generate_key(DH *dh);

int DH_compute_key(unsigned char *key, BIGNUM *pub_key, DH *dh);
```

DESCRIPTION

DH_generate_key() performs the first step of a Diffie-Hellman key exchange by generating private and public DH values. By calling *DH_compute_key()*, these are combined with the other party's public value to compute the shared key.

DH_generate_key() expects **dh** to contain the shared parameters **dh−>p** and **dh−>g**. It generates a random private DH value unless **dh−>priv_key** is already set, and computes the corresponding public value **dh−>pub_key**, which can then be published.

DH_compute_key() computes the shared secret from the private DH value in **dh** and the other party's public value in **pub_key** and stores it in **key**. **key** must point to **DH_size(dh)** bytes of memory.

RETURN VALUES

DH_generate_key() returns 1 on success, 0 otherwise.

DH_compute_key() returns the size of the shared secret on success, −1 on error.

The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

openssl_dh (3), *ERR_get_error* (3), *openssl_rand* (3), *DH_size* (3)

HISTORY

DH_generate_key() and *DH_compute_key()* are available in all versions of SSLeay and OpenSSL.

NAME

DH_generate_parameters, DH_check − generate and check Diffie−Hellman parameters

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dh.h>

DH *DH_generate_parameters(int prime_len, int generator,
    void (*callback)(int, int, void *), void *cb_arg);

int DH_check(DH *dh, int *codes);
```

DESCRIPTION

DH_generate_parameters() generates Diffie-Hellman parameters that can be shared among a group of users, and returns them in a newly allocated **DH** structure. The pseudo-random number generator must be seeded prior to calling *DH_generate_parameters()*.

prime_len is the length in bits of the safe prime to be generated. **generator** is a small number > 1, typically 2 or 5.

A callback function may be used to provide feedback about the progress of the key generation. If **callback** is not **NULL**, it will be called as described in *BN_generate_prime* (3) while a random prime number is generated, and when a prime has been found, **callback(3, 0, cb_arg)** is called.

DH_check() validates Diffie-Hellman parameters. It checks that **p** is a safe prime, and that **g** is a suitable generator. In the case of an error, the bit flags DH_CHECK_P_NOT_SAFE_PRIME or DH_NOT_SUITABLE_GENERATOR are set in ***codes**. DH_UNABLE_TO_CHECK_GENERATOR is set if the generator cannot be checked, i.e. it does not equal 2 or 5.

RETURN VALUES

DH_generate_parameters() returns a pointer to the DH structure, or NULL if the parameter generation fails. The error codes can be obtained by *ERR_get_error* (3).

DH_check() returns 1 if the check could be performed, 0 otherwise.

NOTES

DH_generate_parameters() may run for several hours before finding a suitable prime.

The parameters generated by *DH_generate_parameters()* are not to be used in signature schemes.

BUGS

If **generator** is not 2 or 5, **dh−>g**=**generator** is not a usable generator.

SEE ALSO

openssl_dh (3), *ERR_get_error* (3), *openssl_rand* (3), *DH_free* (3)

HISTORY

DH_check() is available in all versions of SSLeay and OpenSSL. The **cb_arg** argument to *DH_generate_parameters()* was added in SSLeay 0.9.0.

In versions before OpenSSL 0.9.5, DH_CHECK_P_NOT_STRONG_PRIME is used instead of DH_CHECK_P_NOT_SAFE_PRIME.

NAME

DH_get_ex_new_index, DH_set_ex_data, DH_get_ex_data − add application specific data to DH structures

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dh.h>

int DH_get_ex_new_index(long argl, void *argp,
                CRYPTO_EX_new *new_func,
                CRYPTO_EX_dup *dup_func,
                CRYPTO_EX_free *free_func);

int DH_set_ex_data(DH *d, int idx, void *arg);

char *DH_get_ex_data(DH *d, int idx);
```

DESCRIPTION

These functions handle application specific data in DH structures. Their usage is identical to that of *RSA_get_ex_new_index()*, *RSA_set_ex_data()* and *RSA_get_ex_data()* as described in *RSA_get_ex_new_index* (3).

SEE ALSO

RSA_get_ex_new_index (3), *openssl_dh* (3)

HISTORY

DH_get_ex_new_index(), *DH_set_ex_data()* and *DH_get_ex_data()* are available since OpenSSL 0.9.5.

NAME

DH_new, DH_free − allocate and free DH objects

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dh.h>

DH* DH_new(void);

void DH_free(DH *dh);
```

DESCRIPTION

DH_new() allocates and initializes a **DH** structure.

DH_free() frees the **DH** structure and its components. The values are erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, *DH_new()* returns **NULL** and sets an error code that can be obtained by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

DH_free() returns no value.

SEE ALSO

openssl_dh (3), *ERR_get_error* (3), *DH_generate_parameters* (3), *DH_generate_key* (3)

HISTORY

DH_new() and *DH_free()* are available in all versions of SSLeay and OpenSSL.

NAME

DH_set_default_method, DH_get_default_method, DH_set_method, DH_new_method, DH_OpenSSL − select DH method

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dh.h>
#include <openssl/engine.h>

void DH_set_default_method(const DH_METHOD *meth);

const DH_METHOD *DH_get_default_method(void);

int DH_set_method(DH *dh, const DH_METHOD *meth);

DH *DH_new_method(ENGINE *engine);

const DH_METHOD *DH_OpenSSL(void);
```

DESCRIPTION

A **DH_METHOD** specifies the functions that OpenSSL uses for Diffie-Hellman operations. By modifying the method, alternative implementations such as hardware accelerators may be used. IMPORTANT: See the NOTES section for important information about how these DH API functions are affected by the use of **ENGINE** API calls.

Initially, the default DH_METHOD is the OpenSSL internal implementation, as returned by *DH_OpenSSL()*.

DH_set_default_method() makes **meth** the default method for all DH structures created later. **NB**: This is true only whilst no ENGINE has been set as a default for DH, so this function is no longer recommended.

DH_get_default_method() returns a pointer to the current default DH_METHOD. However, the meaningfulness of this result is dependent on whether the ENGINE API is being used, so this function is no longer recommended.

DH_set_method() selects **meth** to perform all operations using the key **dh**. This will replace the DH_METHOD used by the DH key and if the previous method was supplied by an ENGINE, the handle to that ENGINE will be released during the change. It is possible to have DH keys that only work with certain DH_METHOD implementations (eg. from an ENGINE module that supports embedded hardware-protected keys), and in such cases attempting to change the DH_METHOD for the key can have unexpected results.

DH_new_method() allocates and initializes a DH structure so that **engine** will be used for the DH operations. If **engine** is NULL, the default ENGINE for DH operations is used, and if no default ENGINE is set, the DH_METHOD controlled by *DH_set_default_method()* is used.

THE DH_METHOD STRUCTURE

```
typedef struct dh_meth_st
{
    /* name of the implementation */
       const char *name;

    /* generate private and public DH values for key agreement */
       int (*generate_key)(DH *dh);

    /* compute shared secret */
       int (*compute_key)(unsigned char *key, BIGNUM *pub_key, DH *dh);

    /* compute r = a ^ p mod m (May be NULL for some implementations) */
       int (*bn_mod_exp)(DH *dh, BIGNUM *r, BIGNUM *a, const BIGNUM *p,
                              const BIGNUM *m, BN_CTX *ctx,
                              BN_MONT_CTX *m_ctx);
```

```
        /* called at DH_new */
           int (*init)(DH *dh);

        /* called at DH_free */
           int (*finish)(DH *dh);

           int flags;

           char *app_data; /* ?? */

    } DH_METHOD;
```

RETURN VALUES

DH_OpenSSL() and *DH_get_default_method()* return pointers to the respective **DH_METHOD**s.

DH_set_default_method() returns no value.

DH_set_method() returns non-zero if the provided **meth** was successfully set as the method for **dh** (including unloading the ENGINE handle if the previous method was supplied by an ENGINE).

DH_new_method() returns NULL and sets an error code that can be obtained by *ERR_get_error* (3) if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

NOTES

As of version 0.9.7, DH_METHOD implementations are grouped together with other algorithmic APIs (eg. RSA_METHOD, EVP_CIPHER, etc) in **ENGINE** modules. If a default ENGINE is specified for DH functionality using an ENGINE API function, that will override any DH defaults set using the DH API (ie. *DH_set_default_method()*). For this reason, the ENGINE API is the recommended way to control default implementations for use in DH and other cryptographic algorithms.

SEE ALSO

openssl_dh (3), *DH_new* (3)

HISTORY

DH_set_default_method(), *DH_get_default_method()*, *DH_set_method()*, *DH_new_method()* and *DH_OpenSSL()* were added in OpenSSL 0.9.4.

DH_set_default_openssl_method() and *DH_get_default_openssl_method()* replaced *DH_set_default_method()* and *DH_get_default_method()* respectively, and *DH_set_method()* and *DH_new_method()* were altered to use **ENGINE**s rather than **DH_METHOD**s during development of the engine version of OpenSSL 0.9.6. For 0.9.7, the handling of defaults in the ENGINE API was restructured so that this change was reversed, and behaviour of the other functions resembled more closely the previous behaviour. The behaviour of defaults in the ENGINE API now transparently overrides the behaviour of defaults in the DH API without requiring changing these function prototypes.

NAME

DH_size − get Diffie−Hellman prime size

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dh.h>

int DH_size(DH *dh);
```

DESCRIPTION

This function returns the Diffie-Hellman size in bytes. It can be used to determine how much memory must be allocated for the shared secret computed by *DH_compute_key()*.

dh−>p must not be **NULL**.

RETURN VALUE

The size in bytes.

SEE ALSO

openssl_dh (3), *DH_generate_key* (3)

HISTORY

DH_size() is available in all versions of SSLeay and OpenSSL.

NAME

DSA_SIG_new, DSA_SIG_free − allocate and free DSA signature objects

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dsa.h>

DSA_SIG *DSA_SIG_new(void);

void   DSA_SIG_free(DSA_SIG *a);
```

DESCRIPTION

DSA_SIG_new() allocates and initializes a **DSA_SIG** structure.

DSA_SIG_free() frees the **DSA_SIG** structure and its components. The values are erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, *DSA_SIG_new()* returns **NULL** and sets an error code that can be obtained by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

DSA_SIG_free() returns no value.

SEE ALSO

openssl_dsa (3), *ERR_get_error* (3), *DSA_do_sign* (3)

HISTORY

DSA_SIG_new() and *DSA_SIG_free()* were added in OpenSSL 0.9.3.

NAME

DSA_do_sign, DSA_do_verify − raw DSA signature operations

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dsa.h>

DSA_SIG *DSA_do_sign(const unsigned char *dgst, int dlen, DSA *dsa);

int DSA_do_verify(const unsigned char *dgst, int dgst_len,
                  DSA_SIG *sig, DSA *dsa);
```

DESCRIPTION

DSA_do_sign() computes a digital signature on the **len** byte message digest **dgst** using the private key **dsa** and returns it in a newly allocated **DSA_SIG** structure.

DSA_sign_setup (3) may be used to precompute part of the signing operation in case signature generation is time−critical.

DSA_do_verify() verifies that the signature **sig** matches a given message digest **dgst** of size **len**. **dsa** is the signer's public key.

RETURN VALUES

DSA_do_sign() returns the signature, NULL on error. *DSA_do_verify()* returns 1 for a valid signature, 0 for an incorrect signature and −1 on error. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

openssl_dsa (3), *ERR_get_error* (3), *openssl_rand* (3), *DSA_SIG_new* (3), *DSA_sign* (3)

HISTORY

DSA_do_sign() and *DSA_do_verify()* were added in OpenSSL 0.9.3.

NAME

DSA_dup_DH − create a DH structure out of DSA structure

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dsa.h>

DH * DSA_dup_DH(const DSA *r);
```

DESCRIPTION

DSA_dup_DH() duplicates DSA parameters/keys as DH parameters/keys. q is lost during that conversion, but the resulting DH parameters contain its length.

RETURN VALUE

DSA_dup_DH() returns the new **DH** structure, and NULL on error. The error codes can be obtained by *ERR_get_error* (3).

NOTE

Be careful to avoid small subgroup attacks when using this.

SEE ALSO

openssl_dh (3), *openssl_dsa* (3), *ERR_get_error* (3)

HISTORY

DSA_dup_DH() was added in OpenSSL 0.9.4.

NAME

DSA_generate_key − generate DSA key pair

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dsa.h>

int DSA_generate_key(DSA *a);
```

DESCRIPTION

DSA_generate_key() expects **a** to contain DSA parameters. It generates a new key pair and stores it in **a−>pub_key** and **a−>priv_key**.

The PRNG must be seeded prior to calling *DSA_generate_key()*.

RETURN VALUE

DSA_generate_key() returns 1 on success, 0 otherwise. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

openssl_dsa (3), *ERR_get_error* (3), *openssl_rand* (3), *DSA_generate_parameters* (3)

HISTORY

DSA_generate_key() is available since SSLeay 0.8.

NAME

DSA_generate_parameters − generate DSA parameters

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dsa.h>

DSA *DSA_generate_parameters(int bits, unsigned char *seed,
                int seed_len, int *counter_ret, unsigned long *h_ret,
                void (*callback)(int, int, void *), void *cb_arg);
```

DESCRIPTION

DSA_generate_parameters() generates primes p and q and a generator g for use in the DSA.

bits is the length of the prime to be generated; the DSS allows a maximum of 1024 bits.

If **seed** is **NULL** or **seed_len** < 20, the primes will be generated at random. Otherwise, the seed is used to generate them. If the given seed does not yield a prime q, a new random seed is chosen and placed at **seed**.

DSA_generate_parameters() places the iteration count in ***counter_ret** and a counter used for finding a generator in ***h_ret**, unless these are **NULL**.

A callback function may be used to provide feedback about the progress of the key generation. If **callback** is not **NULL**, it will be called as follows:

- When a candidate for q is generated, **callback(0, m++, cb_arg)** is called (m is 0 for the first candidate).
- When a candidate for q has passed a test by trial division, **callback(1, −1, cb_arg)** is called. While a candidate for q is tested by Miller-Rabin primality tests, **callback(1, i, cb_arg)** is called in the outer loop (once for each witness that confirms that the candidate may be prime); i is the loop counter (starting at 0).
- When a prime q has been found, **callback(2, 0, cb_arg)** and **callback(3, 0, cb_arg)** are called.
- Before a candidate for p (other than the first) is generated and tested, **callback(0, counter, cb_arg)** is called.
- When a candidate for p has passed the test by trial division, **callback(1, −1, cb_arg)** is called. While it is tested by the Miller-Rabin primality test, **callback(1, i, cb_arg)** is called in the outer loop (once for each witness that confirms that the candidate may be prime). i is the loop counter (starting at 0).
- When p has been found, **callback(2, 1, cb_arg)** is called.
- When the generator has been found, **callback(3, 1, cb_arg)** is called.

RETURN VALUE

DSA_generate_parameters() returns a pointer to the DSA structure, or **NULL** if the parameter generation fails. The error codes can be obtained by *ERR_get_error* (3).

BUGS

Seed lengths > 20 are not supported.

SEE ALSO

openssl_dsa (3), *ERR_get_error* (3), *openssl_rand* (3), *DSA_free* (3)

HISTORY

DSA_generate_parameters() appeared in SSLeay 0.8. The **cb_arg** argument was added in SSLeay 0.9.0. In versions up to OpenSSL 0.9.4, **callback(1, ...)** was called in the inner loop of the Miller-Rabin test whenever it reached the squaring step (the parameters to **callback** did not reveal how many witnesses had been tested); since OpenSSL 0.9.5, **callback(1, ...)** is called as in *BN_is_prime* (3), i.e. once for each witness. =cut

NAME

DSA_get_ex_new_index, DSA_set_ex_data, DSA_get_ex_data − add application specific data to DSA structures

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/DSA.h>

int DSA_get_ex_new_index(long argl, void *argp,
                CRYPTO_EX_new *new_func,
                CRYPTO_EX_dup *dup_func,
                CRYPTO_EX_free *free_func);

int DSA_set_ex_data(DSA *d, int idx, void *arg);

char *DSA_get_ex_data(DSA *d, int idx);
```

DESCRIPTION

These functions handle application specific data in DSA structures. Their usage is identical to that of *RSA_get_ex_new_index()*, *RSA_set_ex_data()* and *RSA_get_ex_data()* as described in *RSA_get_ex_new_index* (3).

SEE ALSO

RSA_get_ex_new_index (3), *openssl_dsa* (3)

HISTORY

DSA_get_ex_new_index(), *DSA_set_ex_data()* and *DSA_get_ex_data()* are available since OpenSSL 0.9.5.

NAME

DSA_new, DSA_free − allocate and free DSA objects

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dsa.h>

DSA* DSA_new(void);

void DSA_free(DSA *dsa);
```

DESCRIPTION

DSA_new() allocates and initializes a **DSA** structure. It is equivalent to calling DSA_new_method(NULL).

DSA_free() frees the **DSA** structure and its components. The values are erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, *DSA_new()* returns **NULL** and sets an error code that can be obtained by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

DSA_free() returns no value.

SEE ALSO

openssl_dsa (3), *ERR_get_error* (3), *DSA_generate_parameters* (3), *DSA_generate_key* (3)

HISTORY

DSA_new() and *DSA_free()* are available in all versions of SSLeay and OpenSSL.

NAME

DSA_set_default_method, DSA_get_default_method, DSA_set_method, DSA_new_method, DSA_OpenSSL − select DSA method

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dsa.h>
#include <openssl/engine.h>

void DSA_set_default_method(const DSA_METHOD *meth);

const DSA_METHOD *DSA_get_default_method(void);

int DSA_set_method(DSA *dsa, const DSA_METHOD *meth);

DSA *DSA_new_method(ENGINE *engine);

DSA_METHOD *DSA_OpenSSL(void);
```

DESCRIPTION

A **DSA_METHOD** specifies the functions that OpenSSL uses for DSA operations. By modifying the method, alternative implementations such as hardware accelerators may be used. IMPORTANT: See the NOTES section for important information about how these DSA API functions are affected by the use of **ENGINE** API calls.

Initially, the default DSA_METHOD is the OpenSSL internal implementation, as returned by *DSA_OpenSSL()*.

DSA_set_default_method() makes **meth** the default method for all DSA structures created later. **NB**: This is true only whilst no ENGINE has been set as a default for DSA, so this function is no longer recommended.

DSA_get_default_method() returns a pointer to the current default DSA_METHOD. However, the meaningfulness of this result is dependent on whether the ENGINE API is being used, so this function is no longer recommended.

DSA_set_method() selects **meth** to perform all operations using the key **rsa**. This will replace the DSA_METHOD used by the DSA key and if the previous method was supplied by an ENGINE, the handle to that ENGINE will be released during the change. It is possible to have DSA keys that only work with certain DSA_METHOD implementations (eg. from an ENGINE module that supports embedded hardware-protected keys), and in such cases attempting to change the DSA_METHOD for the key can have unexpected results.

DSA_new_method() allocates and initializes a DSA structure so that **engine** will be used for the DSA operations. If **engine** is NULL, the default engine for DSA operations is used, and if no default ENGINE is set, the DSA_METHOD controlled by *DSA_set_default_method()* is used.

THE DSA_METHOD STRUCTURE

```
struct
 {
   /* name of the implementation */
     const char *name;

     /* sign */
        DSA_SIG *(*dsa_do_sign)(const unsigned char *dgst, int dlen,
                                DSA *dsa);

     /* pre-compute kˆ-1 and r */
        int (*dsa_sign_setup)(DSA *dsa, BN_CTX *ctx_in, BIGNUM **kinvp,
                                BIGNUM **rp);

     /* verify */
        int (*dsa_do_verify)(const unsigned char *dgst, int dgst_len,
                                DSA_SIG *sig, DSA *dsa);
```

```
        /* compute rr = a1^p1 * a2^p2 mod m (May be NULL for some
                                             implementations) */
           int (*dsa_mod_exp)(DSA *dsa, BIGNUM *rr, BIGNUM *a1, BIGNUM *p1,
                              BIGNUM *a2, BIGNUM *p2, BIGNUM *m,
                              BN_CTX *ctx, BN_MONT_CTX *in_mont);

        /* compute r = a ^ p mod m (May be NULL for some implementations) */
           int (*bn_mod_exp)(DSA *dsa, BIGNUM *r, BIGNUM *a,
                              const BIGNUM *p, const BIGNUM *m,
                              BN_CTX *ctx, BN_MONT_CTX *m_ctx);

        /* called at DSA_new */
           int (*init)(DSA *DSA);

        /* called at DSA_free */
           int (*finish)(DSA *DSA);

           int flags;

           char *app_data; /* ?? */

    } DSA_METHOD;
```

RETURN VALUES

DSA_OpenSSL() and *DSA_get_default_method()* return pointers to the respective **DSA_METHOD**s.

DSA_set_default_method() returns no value.

DSA_set_method() returns non-zero if the provided **meth** was successfully set as the method for **dsa** (including unloading the ENGINE handle if the previous method was supplied by an ENGINE).

DSA_new_method() returns NULL and sets an error code that can be obtained by *ERR_get_error* (3) if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

NOTES

As of version 0.9.7, DSA_METHOD implementations are grouped together with other algorithmic APIs (eg. RSA_METHOD, EVP_CIPHER, etc) in **ENGINE** modules. If a default ENGINE is specified for DSA functionality using an ENGINE API function, that will override any DSA defaults set using the DSA API (ie. *DSA_set_default_method()*). For this reason, the ENGINE API is the recommended way to control default implementations for use in DSA and other cryptographic algorithms.

SEE ALSO

openssl_dsa (3), *DSA_new* (3)

HISTORY

DSA_set_default_method(), *DSA_get_default_method()*, *DSA_set_method()*, *DSA_new_method()* and *DSA_OpenSSL()* were added in OpenSSL 0.9.4.

DSA_set_default_openssl_method() and *DSA_get_default_openssl_method()* replaced *DSA_set_default_method()* and *DSA_get_default_method()* respectively, and *DSA_set_method()* and *DSA_new_method()* were altered to use **ENGINE**s rather than **DSA_METHOD**s during development of the engine version of OpenSSL 0.9.6. For 0.9.7, the handling of defaults in the ENGINE API was restructured so that this change was reversed, and behaviour of the other functions resembled more closely the previous behaviour. The behaviour of defaults in the ENGINE API now transparently overrides the behaviour of defaults in the DSA API without requiring changing these function prototypes.

NAME

DSA_sign, DSA_sign_setup, DSA_verify − DSA signatures

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dsa.h>

int    DSA_sign(int type, const unsigned char *dgst, int len,
                unsigned char *sigret, unsigned int *siglen, DSA *dsa);

int    DSA_sign_setup(DSA *dsa, BN_CTX *ctx, BIGNUM **kinvp,
                BIGNUM **rp);

int    DSA_verify(int type, const unsigned char *dgst, int len,
                unsigned char *sigbuf, int siglen, DSA *dsa);
```

DESCRIPTION

DSA_sign() computes a digital signature on the **len** byte message digest **dgst** using the private key **dsa** and places its ASN.1 DER encoding at **sigret**. The length of the signature is places in ***siglen**. **sigret** must point to DSA_size(**dsa**) bytes of memory.

DSA_sign_setup() may be used to precompute part of the signing operation in case signature generation is time−critical. It expects **dsa** to contain DSA parameters. It places the precomputed values in newly allocated **BIGNUM**s at ***kinvp** and ***rp**, after freeing the old ones unless ***kinvp** and ***rp** are NULL. These values may be passed to *DSA_sign()* in **dsa−>kinv** and **dsa−>r**. **ctx** is a pre-allocated **BN_CTX** or NULL.

DSA_verify() verifies that the signature **sigbuf** of size **siglen** matches a given message digest **dgst** of size **len**. **dsa** is the signer's public key.

The **type** parameter is ignored.

The PRNG must be seeded before *DSA_sign()* (or *DSA_sign_setup()*) is called.

RETURN VALUES

DSA_sign() and *DSA_sign_setup()* return 1 on success, 0 on error. *DSA_verify()* returns 1 for a valid signature, 0 for an incorrect signature and −1 on error. The error codes can be obtained by *ERR_get_error* (3).

CONFORMING TO

US Federal Information Processing Standard FIPS 186 (Digital Signature Standard, DSS), ANSI X9.30

SEE ALSO

openssl_dsa (3), *ERR_get_error* (3), *openssl_rand* (3), *DSA_do_sign* (3)

HISTORY

DSA_sign() and *DSA_verify()* are available in all versions of SSLeay. *DSA_sign_setup()* was added in SSLeay 0.8.

NAME

DSA_size − get DSA signature size

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dsa.h>

int DSA_size(const DSA *dsa);
```

DESCRIPTION

This function returns the size of an ASN.1 encoded DSA signature in bytes. It can be used to determine how much memory must be allocated for a DSA signature.

dsa->q must not be **NULL**.

RETURN VALUE

The size in bytes.

SEE ALSO

openssl_dsa (3), *DSA_sign* (3)

HISTORY

DSA_size() is available in all versions of SSLeay and OpenSSL.

NAME

ERR_GET_LIB, ERR_GET_FUNC, ERR_GET_REASON − get library, function and reason code

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/err.h>

int ERR_GET_LIB(unsigned long e);

int ERR_GET_FUNC(unsigned long e);

int ERR_GET_REASON(unsigned long e);
```

DESCRIPTION

The error code returned by *ERR_get_error()* consists of a library number, function code and reason code. *ERR_GET_LIB()*, *ERR_GET_FUNC()* and *ERR_GET_REASON()* can be used to extract these.

The library number and function code describe where the error occurred, the reason code is the information about what went wrong.

Each sub-library of OpenSSL has a unique library number; function and reason codes are unique within each sub−library. Note that different libraries may use the same value to signal different functions and reasons.

ERR_R_... reason codes such as **ERR_R_MALLOC_FAILURE** are globally unique. However, when checking for sub-library specific reason codes, be sure to also compare the library number.

ERR_GET_LIB(), *ERR_GET_FUNC()* and *ERR_GET_REASON()* are macros.

RETURN VALUES

The library number, function code and reason code respectively.

SEE ALSO

openssl_err (3), *ERR_get_error* (3)

HISTORY

ERR_GET_LIB(), *ERR_GET_FUNC()* and *ERR_GET_REASON()* are available in all versions of SSLeay and OpenSSL.

NAME

ERR_clear_error − clear the error queue

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/err.h>

void ERR_clear_error(void);
```

DESCRIPTION

ERR_clear_error() empties the current thread’s error queue.

RETURN VALUES

ERR_clear_error() has no return value.

SEE ALSO

openssl_err (3), *ERR_get_error* (3)

HISTORY

ERR_clear_error() is available in all versions of SSLeay and OpenSSL.

NAME

ERR_error_string, ERR_error_string_n, ERR_lib_error_string, ERR_func_error_string, ERR_reason_error_string − obtain human−readable error message

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/err.h>

char *ERR_error_string(unsigned long e, char *buf);
void ERR_error_string_n(unsigned long e, char *buf, size_t len);

const char *ERR_lib_error_string(unsigned long e);
const char *ERR_func_error_string(unsigned long e);
const char *ERR_reason_error_string(unsigned long e);
```

DESCRIPTION

ERR_error_string() generates a human-readable string representing the error code *e*, and places it at *buf*. *buf* must be at least 120 bytes long. If *buf* is **NULL**, the error string is placed in a static buffer. *ERR_error_string_n()* is a variant of *ERR_error_string()* that writes at most *len* characters (including the terminating 0) and truncates the string if necessary. For *ERR_error_string_n()*, *buf* may not be **NULL**.

The string will have the following format:

```
error:[error code]:[library name]:[function name]:[reason string]
```

error code is an 8 digit hexadecimal number, *library name*, *function name* and *reason string* are ASCII text.

ERR_lib_error_string(), *ERR_func_error_string()* and *ERR_reason_error_string()* return the library name, function name and reason string respectively.

The OpenSSL error strings should be loaded by calling *ERR_load_crypto_strings*(3) or, for SSL applications, *SSL_load_error_strings*(3) first. If there is no text string registered for the given error code, the error string will contain the numeric code.

ERR_print_errors(3) can be used to print all error codes currently in the queue.

RETURN VALUES

ERR_error_string() returns a pointer to a static buffer containing the string if *buf* == **NULL**, *buf* otherwise.

ERR_lib_error_string(), *ERR_func_error_string()* and *ERR_reason_error_string()* return the strings, and **NULL** if none is registered for the error code.

SEE ALSO

openssl_err(3), *ERR_get_error*(3), *ERR_load_crypto_strings*(3), *SSL_load_error_strings*(3) *ERR_print_errors*(3)

HISTORY

ERR_error_string() is available in all versions of SSLeay and OpenSSL. *ERR_error_string_n()* was added in OpenSSL 0.9.6.

NAME

ERR_get_error, ERR_peek_error, ERR_peek_last_error, ERR_get_error_line, ERR_peek_error_line, ERR_peek_last_error_line, ERR_get_error_line_data, ERR_peek_error_line_data, ERR_peek_last_error_line_data − obtain error code and data

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/err.h>

unsigned long ERR_get_error(void);
unsigned long ERR_peek_error(void);
unsigned long ERR_peek_last_error(void);

unsigned long ERR_get_error_line(const char **file, int *line);
unsigned long ERR_peek_error_line(const char **file, int *line);
unsigned long ERR_peek_last_error_line(const char **file, int *line);

unsigned long ERR_get_error_line_data(const char **file, int *line,
        const char **data, int *flags);
unsigned long ERR_peek_error_line_data(const char **file, int *line,
        const char **data, int *flags);
unsigned long ERR_peek_last_error_line_data(const char **file, int *line,
        const char **data, int *flags);
```

DESCRIPTION

ERR_get_error() returns the earliest error code from the thread's error queue and removes the entry. This function can be called repeatedly until there are no more error codes to return.

ERR_peek_error() returns the earliest error code from the thread's error queue without modifying it.

ERR_peek_last_error() returns the latest error code from the thread's error queue without modifying it.

See *ERR_GET_LIB* (3) for obtaining information about location and reason of the error, and *ERR_error_string* (3) for human-readable error messages.

ERR_get_error_line(), *ERR_peek_error_line()* and *ERR_peek_last_error_line()* are the same as the above, but they additionally store the file name and line number where the error occurred in ***file** and ***line**, unless these are **NULL**.

ERR_get_error_line_data(), *ERR_peek_error_line_data()* and *ERR_get_last_error_line_data()* store additional data and flags associated with the error code in ***data** and ***flags**, unless these are **NULL**. ***data** contains a string if ***flags**&**ERR_TXT_STRING**. If it has been allocated by *OPENSSL_malloc()*, ***flags**&**ERR_TXT_MALLOCED** is true.

RETURN VALUES

The error code, or 0 if there is no error in the queue.

SEE ALSO

openssl_err (3), *ERR_error_string* (3), *ERR_GET_LIB* (3)

HISTORY

ERR_get_error(), *ERR_peek_error()*, *ERR_get_error_line()* and *ERR_peek_error_line()* are available in all versions of SSLeay and OpenSSL. *ERR_get_error_line_data()* and *ERR_peek_error_line_data()* were added in SSLeay 0.9.0. *ERR_peek_last_error()*, *ERR_peek_last_error_line()* and *ERR_peek_last_error_line_data()* were added in OpenSSL 0.9.7.

NAME

ERR_load_crypto_strings, SSL_load_error_strings, ERR_free_strings − load and free error strings

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/err.h>

void ERR_load_crypto_strings(void);
void ERR_free_strings(void);

#include <openssl/ssl.h>

void SSL_load_error_strings(void);
```

DESCRIPTION

ERR_load_crypto_strings() registers the error strings for all **libcrypto** functions. *SSL_load_error_strings()* does the same, but also registers the **libssl** error strings.

One of these functions should be called before generating textual error messages. However, this is not required when memory usage is an issue.

ERR_free_strings() frees all previously loaded error strings.

RETURN VALUES

ERR_load_crypto_strings(), *SSL_load_error_strings()* and *ERR_free_strings()* return no values.

SEE ALSO

openssl_err (3), *ERR_error_string* (3)

HISTORY

ERR_load_error_strings(), *SSL_load_error_strings()* and *ERR_free_strings()* are available in all versions of SSLeay and OpenSSL.

NAME

ERR_load_strings, ERR_PACK, ERR_get_next_error_library − load arbitrary error strings

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/err.h>

void ERR_load_strings(int lib, ERR_STRING_DATA str[]);

int ERR_get_next_error_library(void);

unsigned long ERR_PACK(int lib, int func, int reason);
```

DESCRIPTION

ERR_load_strings() registers error strings for library number **lib**.

str is an array of error string data:

```
typedef struct ERR_string_data_st
{
        unsigned long error;
        char *string;
} ERR_STRING_DATA;
```

The error code is generated from the library number and a function and reason code: **error** = ERR_PACK(**lib**, **func**, **reason**). *ERR_PACK()* is a macro.

The last entry in the array is {0,0}.

ERR_get_next_error_library() can be used to assign library numbers to user libraries at runtime.

RETURN VALUE

ERR_load_strings() returns no value. *ERR_PACK()* return the error code. *ERR_get_next_error_library()* returns a new library number.

SEE ALSO

openssl_err (3), *ERR_load_strings* (3)

HISTORY

ERR_load_error_strings() and *ERR_PACK()* are available in all versions of SSLeay and OpenSSL. *ERR_get_next_error_library()* was added in SSLeay 0.9.0.

NAME

ERR_print_errors, ERR_print_errors_fp − print error messages

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/err.h>

void ERR_print_errors(BIO *bp);
void ERR_print_errors_fp(FILE *fp);
```

DESCRIPTION

ERR_print_errors() is a convenience function that prints the error strings for all errors that OpenSSL has recorded to **bp**, thus emptying the error queue.

ERR_print_errors_fp() is the same, except that the output goes to a **FILE**.

The error strings will have the following format:

```
[pid]:error:[error code]:[library name]:[function name]:[reason string]:[file name]
```

error code is an 8 digit hexadecimal number. *library name*, *function name* and *reason string* are ASCII text, as is *optional text message* if one was set for the respective error code.

If there is no text string registered for the given error code, the error string will contain the numeric code.

RETURN VALUES

ERR_print_errors() and *ERR_print_errors_fp()* return no values.

SEE ALSO

openssl_err (3), *ERR_error_string* (3), *ERR_get_error* (3), *ERR_load_crypto_strings* (3), *SSL_load_error_strings* (3)

HISTORY

ERR_print_errors() and *ERR_print_errors_fp()* are available in all versions of SSLeay and OpenSSL.

NAME

ERR_put_error, ERR_add_error_data − record an error

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/err.h>

void ERR_put_error(int lib, int func, int reason, const char *file,
        int line);

void ERR_add_error_data(int num, ...);
```

DESCRIPTION

ERR_put_error() adds an error code to the thread's error queue. It signals that the error of reason code **reason** occurred in function **func** of library **lib**, in line number **line** of **file**. This function is usually called by a macro.

ERR_add_error_data() associates the concatenation of its **num** string arguments with the error code added last.

ERR_load_strings (3) can be used to register error strings so that the application can a generate human-readable error messages for the error code.

RETURN VALUES

ERR_put_error() and *ERR_add_error_data()* return no values.

SEE ALSO

openssl_err (3), *ERR_load_strings* (3)

HISTORY

ERR_put_error() is available in all versions of SSLeay and OpenSSL. *ERR_add_error_data()* was added in SSLeay 0.9.0.

NAME

ERR_remove_state − free a thread’s error queue

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/err.h>

void ERR_remove_state(unsigned long pid);
```

DESCRIPTION

ERR_remove_state() frees the error queue associated with thread **pid**. If **pid** == 0, the current thread will have its error queue removed.

Since error queue data structures are allocated automatically for new threads, they must be freed when threads are terminated in order to avoid memory leaks.

RETURN VALUE

ERR_remove_state() returns no value.

SEE ALSO

openssl_err (3)

HISTORY

ERR_remove_state() is available in all versions of SSLeay and OpenSSL.

NAME

ERR_set_mark, ERR_pop_to_mark − set marks and pop errors until mark

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/err.h>

int ERR_set_mark(void);

int ERR_pop_to_mark(void);
```

DESCRIPTION

ERR_set_mark() sets a mark on the current topmost error record if there is one.

ERR_pop_to_mark() will pop the top of the error stack until a mark is found. The mark is then removed. If there is no mark, the whole stack is removed.

RETURN VALUES

ERR_set_mark() returns 0 if the error stack is empty, otherwise 1.

ERR_pop_to_mark() returns 0 if there was no mark in the error stack, which implies that the stack became empty, otherwise 1.

SEE ALSO

openssl_err (3)

HISTORY

ERR_set_mark() and *ERR_pop_to_mark()* were added in OpenSSL 0.9.8.

NAME

EVP_BytesToKey − password based encryption routine

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>

int EVP_BytesToKey(const EVP_CIPHER *type,const EVP_MD *md,
                   const unsigned char *salt,
                   const unsigned char *data, int datal, int count,
                   unsigned char *key,unsigned char *iv);
```

DESCRIPTION

EVP_BytesToKey() derives a key and IV from various parameters. **type** is the cipher to derive the key and IV for. **md** is the message digest to use. The **salt** paramter is used as a salt in the derivation: it should point to an 8 byte buffer or NULL if no salt is used. **data** is a buffer containing **datal** bytes which is used to derive the keying data. **count** is the iteration count to use. The derived key and IV will be written to **key** and **iv** respectively.

NOTES

A typical application of this function is to derive keying material for an encryption algorithm from a password in the **data** parameter.

Increasing the **count** parameter slows down the algorithm which makes it harder for an attacker to peform a brute force attack using a large number of candidate passwords.

If the total key and IV length is less than the digest length and **MD5** is used then the derivation algorithm is compatible with PKCS#5 v1.5 otherwise a non standard extension is used to derive the extra data.

Newer applications should use more standard algorithms such as PKCS#5 v2.0 for key derivation.

KEY DERIVATION ALGORITHM

The key and IV is derived by concatenating D_1, D_2, etc until enough data is available for the key and IV. D_i is defined as:

```
D_i = HASHˆcount(D_(i-1) || data || salt)
```

where || denotes concatentaion, D_0 is empty, HASH is the digest algorithm in use, HASHˆ1(data) is simply HASH(data), HASHˆ2(data) is HASH(HASH(data)) and so on.

The initial bytes are used for the key and the subsequent bytes for the IV.

RETURN VALUES

EVP_BytesToKey() returns the size of the derived key in bytes.

SEE ALSO

openssl_evp (3), *openssl_rand* (3), *EVP_EncryptInit* (3)

HISTORY

NAME

EVP_MD_CTX_init, EVP_MD_CTX_create, EVP_DigestInit_ex, EVP_DigestUpdate, EVP_DigestFinal_ex, EVP_MD_CTX_cleanup, EVP_MD_CTX_destroy, EVP_MAX_MD_SIZE, EVP_MD_CTX_copy_ex, EVP_MD_CTX_copy, EVP_MD_type, EVP_MD_pkey_type, EVP_MD_size, EVP_MD_block_size, EVP_MD_CTX_md, EVP_MD_CTX_size, EVP_MD_CTX_block_size, EVP_MD_CTX_type, EVP_md_null, EVP_md2, EVP_md5, EVP_sha, EVP_sha1, EVP_dss, EVP_dss1, EVP_mdc2, EVP_ripemd160, EVP_get_digestbyname, EVP_get_digestbynid, EVP_get_digestbyobj − EVP digest routines

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>

void EVP_MD_CTX_init(EVP_MD_CTX *ctx);
EVP_MD_CTX *EVP_MD_CTX_create(void);

int EVP_DigestInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int EVP_DigestUpdate(EVP_MD_CTX *ctx, const void *d, size_t cnt);
int EVP_DigestFinal_ex(EVP_MD_CTX *ctx, unsigned char *md,
       unsigned int *s);

int EVP_MD_CTX_cleanup(EVP_MD_CTX *ctx);
void EVP_MD_CTX_destroy(EVP_MD_CTX *ctx);

int EVP_MD_CTX_copy_ex(EVP_MD_CTX *out,const EVP_MD_CTX *in);

int EVP_DigestInit(EVP_MD_CTX *ctx, const EVP_MD *type);
int EVP_DigestFinal(EVP_MD_CTX *ctx, unsigned char *md,
       unsigned int *s);

int EVP_MD_CTX_copy(EVP_MD_CTX *out,EVP_MD_CTX *in);

#define EVP_MAX_MD_SIZE (16+20) /* The SSLv3 md5+sha1 type */

#define EVP_MD_type(e)                  ((e)->type)
#define EVP_MD_pkey_type(e)             ((e)->pkey_type)
#define EVP_MD_size(e)                  ((e)->md_size)
#define EVP_MD_block_size(e)            ((e)->block_size)

#define EVP_MD_CTX_md(e)                (e)->digest)
#define EVP_MD_CTX_size(e)              EVP_MD_size((e)->digest)
#define EVP_MD_CTX_block_size(e)        EVP_MD_block_size((e)->digest)
#define EVP_MD_CTX_type(e)              EVP_MD_type((e)->digest)

const EVP_MD *EVP_md_null(void);
const EVP_MD *EVP_md2(void);
const EVP_MD *EVP_md5(void);
const EVP_MD *EVP_sha(void);
const EVP_MD *EVP_sha1(void);
const EVP_MD *EVP_dss(void);
const EVP_MD *EVP_dss1(void);
const EVP_MD *EVP_mdc2(void);
const EVP_MD *EVP_ripemd160(void);

const EVP_MD *EVP_get_digestbyname(const char *name);
#define EVP_get_digestbynid(a) EVP_get_digestbyname(OBJ_nid2sn(a))
#define EVP_get_digestbyobj(a) EVP_get_digestbynid(OBJ_obj2nid(a))
```

DESCRIPTION

The EVP digest routines are a high level interface to message digests.

EVP_MD_CTX_init() initializes digest contet **ctx**.

EVP_MD_CTX_create() allocates, initializes and returns a digest contet.

EVP_DigestInit_ex() sets up digest context **ctx** to use a digest **type** from ENGINE **impl**. **ctx** must be initialized before calling this function. **type** will typically be supplied by a functionsuch as *EVP_sha1()*. If **impl** is NULL then the default implementation of digest **type** is used.

EVP_DigestUpdate() hashes **cnt** bytes of data at **d** into the digest context **ctx**. This function can be called several times on the same **ctx** to hash additional data.

EVP_DigestFinal_ex() retrieves the digest value from **ctx** and places it in **md**. If the **s** parameter is not NULL then the number of bytes of data written (i.e. the length of the digest) will be written to the integer at **s**, at most **EVP_MAX_MD_SIZE** bytes will be written. After calling *EVP_DigestFinal_ex()* no additional calls to *EVP_DigestUpdate()* can be made, but *EVP_DigestInit_ex()* can be called to initialize a new digest operation.

EVP_MD_CTX_cleanup() cleans up digest context **ctx**, it should be called after a digest context is no longer needed.

EVP_MD_CTX_destroy() cleans up digest context **ctx** and frees up the space allocated to it, it should be called only on a context created using *EVP_MD_CTX_create()*.

EVP_MD_CTX_copy_ex() can be used to copy the message digest state from **in** to **out**. This is useful if large amounts of data are to be hashed which only differ in the last few bytes. **out** must be initialized before calling this function.

EVP_DigestInit() behaves in the same way as *EVP_DigestInit_ex()* except the passed context **ctx** does not have to be initialized, and it always uses the default digest implementation.

EVP_DigestFinal() is similar to *EVP_DigestFinal_ex()* except the digest contet **ctx** is automatically cleaned up.

EVP_MD_CTX_copy() is similar to *EVP_MD_CTX_copy_ex()* except the destination **out** does not have to be initialized.

EVP_MD_size() and *EVP_MD_CTX_size()* return the size of the message digest when passed an **EVP_MD** or an **EVP_MD_CTX** structure, i.e. the size of the hash.

EVP_MD_block_size() and *EVP_MD_CTX_block_size()* return the block size of the message digest when passed an **EVP_MD** or an **EVP_MD_CTX** structure.

EVP_MD_type() and *EVP_MD_CTX_type()* return the NID of the OBJECT IDENTIFIER representing the given message digest when passed an **EVP_MD** structure. For example EVP_MD_type(*EVP_sha1()*) returns **NID_sha1**. This function is normally used when setting ASN1 OIDs.

EVP_MD_CTX_md() returns the **EVP_MD** structure corresponding to the passed **EVP_MD_CTX**.

EVP_MD_pkey_type() returns the NID of the public key signing algorithm associated with this digest. For example *EVP_sha1()* is associated with RSA so this will return **NID_sha1WithRSAEncryption**. This ''link'' between digests and signature algorithms may not be retained in future versions of OpenSSL.

EVP_md2(), *EVP_md5()*, *EVP_sha()*, *EVP_sha1()*, *EVP_mdc2()* and *EVP_ripemd160()* return **EVP_MD** structures for the MD2, MD5, SHA, SHA1, MDC2 and RIPEMD160 digest algorithms respectively. The associated signature algorithm is RSA in each case.

EVP_dss() and *EVP_dss1()* return **EVP_MD** structures for SHA and SHA1 digest algorithms but using DSS (DSA) for the signature algorithm.

EVP_md_null() is a ''null'' message digest that does nothing: i.e. the hash it returns is of zero length.

EVP_get_digestbyname(), *EVP_get_digestbynid()* and *EVP_get_digestbyobj()* return an **EVP_MD** structure when passed a digest name, a digest NID or an ASN1_OBJECT structure respectively. The digest table must be initialized using, for example, *OpenSSL_add_all_digests()* for these functions to work.

RETURN VALUES

EVP_DigestInit_ex(), *EVP_DigestUpdate()* and *EVP_DigestFinal_ex()* return 1 for success and 0 for failure.

EVP_MD_CTX_copy_ex() returns 1 if successful or 0 for failure.

EVP_MD_type(), *EVP_MD_pkey_type()* and *EVP_MD_type()* return the NID of the corresponding OBJECT IDENTIFIER or NID_undef if none exists.

EVP_MD_size(), *EVP_MD_block_size()*, EVP_MD_CTX_size(e), *EVP_MD_size()*, *EVP_MD_CTX_block_size()* and *EVP_MD_block_size()* return the digest or block size in bytes.

EVP_md_null(), *EVP_md2()*, *EVP_md5()*, *EVP_sha()*, *EVP_sha1()*, *EVP_dss()*, *EVP_dss1()*, *EVP_mdc2()* and *EVP_ripemd160()* return pointers to the corresponding EVP_MD structures.

EVP_get_digestbyname(), *EVP_get_digestbynid()* and *EVP_get_digestbyobj()* return either an **EVP_MD** structure or NULL if an error occurs.

NOTES

The **EVP** interface to message digests should almost always be used in preference to the low level interfaces. This is because the code then becomes transparent to the digest used and much more flexible.

SHA1 is the digest of choice for new applications. The other digest algorithms are still in common use.

For most applications the **impl** parameter to *EVP_DigestInit_ex()* will be set to NULL to use the default digest implementation.

The functions *EVP_DigestInit()*, *EVP_DigestFinal()* and *EVP_MD_CTX_copy()* are obsolete but are retained to maintain compatibility with existing code. New applications should use *EVP_DigestInit_ex()*, *EVP_DigestFinal_ex()* and *EVP_MD_CTX_copy_ex()* because they can efficiently reuse a digest context instead of initializing and cleaning it up on each call and allow non default implementations of digests to be specified.

In OpenSSL 0.9.7 and later if digest contexts are not cleaned up after use memory leaks will occur.

EXAMPLE

This example digests the data ‘‘Test Message\n’’ and ‘‘Hello World\n’’, using the digest name passed on the command line.

```
#include <stdio.h>
#include <openssl/evp.h>

main(int argc, char *argv[])
{
EVP_MD_CTX mdctx;
const EVP_MD *md;
char mess1[] = "Test Message\n";
char mess2[] = "Hello World\n";
unsigned char md_value[EVP_MAX_MD_SIZE];
int md_len, i;

OpenSSL_add_all_digests();

if(!argv[1]) {
       printf("Usage: mdtest digestname\n");
       exit(1);
}

md = EVP_get_digestbyname(argv[1]);

if(!md) {
       printf("Unknown message digest %s\n", argv[1]);
       exit(1);
}
```

```
EVP_MD_CTX_init(&mdctx);
EVP_DigestInit_ex(&mdctx, md, NULL);
EVP_DigestUpdate(&mdctx, mess1, strlen(mess1));
EVP_DigestUpdate(&mdctx, mess2, strlen(mess2));
EVP_DigestFinal_ex(&mdctx, md_value, &md_len);
EVP_MD_CTX_cleanup(&mdctx);

printf("Digest is: ");
for(i = 0; i < md_len; i++) printf("%02x", md_value[i]);
printf("\n");
}
```

BUGS

The link between digests and signing algorithms results in a situation where *EVP_sha1()* must be used with RSA and *EVP_dss1()* must be used with DSS even though they are identical digests.

SEE ALSO

openssl_evp (3), *openssl_hmac* (3), *md2* (3), *openssl_md5* (3), *openssl_mdc2* (3), *openssl_ripemd* (3), *openssl_sha* (3), *openssl_dgst* (1)

HISTORY

EVP_DigestInit(), *EVP_DigestUpdate()* and *EVP_DigestFinal()* are available in all versions of SSLeay and OpenSSL.

EVP_MD_CTX_init(), *EVP_MD_CTX_create()*, *EVP_MD_CTX_copy_ex()*, *EVP_MD_CTX_cleanup()*, *EVP_MD_CTX_destroy()*, *EVP_DigestInit_ex()* and *EVP_DigestFinal_ex()* were added in OpenSSL 0.9.7.

EVP_md_null(), *EVP_md2()*, *EVP_md5()*, *EVP_sha()*, *EVP_sha1()*, *EVP_dss()*, *EVP_dss1()*, *EVP_mdc2()* and *EVP_ripemd160()* were changed to return truely const EVP_MD * in OpenSSL 0.9.7.

NAME

EVP_CIPHER_CTX_init, EVP_EncryptInit_ex, EVP_EncryptUpdate, EVP_EncryptFinal_ex, EVP_DecryptInit_ex, EVP_DecryptUpdate, EVP_DecryptFinal_ex, EVP_CipherInit_ex, EVP_CipherUpdate, EVP_CipherFinal_ex, EVP_CIPHER_CTX_set_key_length, EVP_CIPHER_CTX_ctrl, EVP_CIPHER_CTX_cleanup, EVP_EncryptInit, EVP_EncryptFinal, EVP_DecryptInit, EVP_DecryptFinal, EVP_CipherInit, EVP_CipherFinal, EVP_get_cipherbyname, EVP_get_cipherbynid, EVP_get_cipherbyobj, EVP_CIPHER_nid, EVP_CIPHER_block_size, EVP_CIPHER_key_length, EVP_CIPHER_iv_length, EVP_CIPHER_flags, EVP_CIPHER_mode, EVP_CIPHER_type, EVP_CIPHER_CTX_cipher, EVP_CIPHER_CTX_nid, EVP_CIPHER_CTX_block_size, EVP_CIPHER_CTX_key_length, EVP_CIPHER_CTX_iv_length, EVP_CIPHER_CTX_get_app_data, EVP_CIPHER_CTX_set_app_data, EVP_CIPHER_CTX_type, EVP_CIPHER_CTX_flags, EVP_CIPHER_CTX_mode, EVP_CIPHER_param_to_asn1, EVP_CIPHER_asn1_to_param, EVP_CIPHER_CTX_set_padding − EVP cipher routines

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>

void EVP_CIPHER_CTX_init(EVP_CIPHER_CTX *a);

int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        ENGINE *impl, unsigned char *key, unsigned char *iv);
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl, unsigned char *in, int inl);
int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl);

int EVP_DecryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        ENGINE *impl, unsigned char *key, unsigned char *iv);
int EVP_DecryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl, unsigned char *in, int inl);
int EVP_DecryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm,
        int *outl);

int EVP_CipherInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        ENGINE *impl, unsigned char *key, unsigned char *iv, int enc);
int EVP_CipherUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl, unsigned char *in, int inl);
int EVP_CipherFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm,
        int *outl);

int EVP_EncryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        unsigned char *key, unsigned char *iv);
int EVP_EncryptFinal(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl);

int EVP_DecryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        unsigned char *key, unsigned char *iv);
int EVP_DecryptFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm,
        int *outl);

int EVP_CipherInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        unsigned char *key, unsigned char *iv, int enc);
int EVP_CipherFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm,
        int *outl);
```

```
int EVP_CIPHER_CTX_set_padding(EVP_CIPHER_CTX *x, int padding);
int EVP_CIPHER_CTX_set_key_length(EVP_CIPHER_CTX *x, int keylen);
int EVP_CIPHER_CTX_ctrl(EVP_CIPHER_CTX *ctx, int type, int arg, void *ptr);
int EVP_CIPHER_CTX_cleanup(EVP_CIPHER_CTX *a);

const EVP_CIPHER *EVP_get_cipherbyname(const char *name);
#define EVP_get_cipherbynid(a) EVP_get_cipherbyname(OBJ_nid2sn(a))
#define EVP_get_cipherbyobj(a) EVP_get_cipherbynid(OBJ_obj2nid(a))

#define EVP_CIPHER_nid(e)             ((e)->nid)
#define EVP_CIPHER_block_size(e)      ((e)->block_size)
#define EVP_CIPHER_key_length(e)      ((e)->key_len)
#define EVP_CIPHER_iv_length(e)               ((e)->iv_len)
#define EVP_CIPHER_flags(e)           ((e)->flags)
#define EVP_CIPHER_mode(e)            ((e)->flags) & EVP_CIPH_MODE)
int EVP_CIPHER_type(const EVP_CIPHER *ctx);

#define EVP_CIPHER_CTX_cipher(e)      ((e)->cipher)
#define EVP_CIPHER_CTX_nid(e)         ((e)->cipher->nid)
#define EVP_CIPHER_CTX_block_size(e)  ((e)->cipher->block_size)
#define EVP_CIPHER_CTX_key_length(e)  ((e)->key_len)
#define EVP_CIPHER_CTX_iv_length(e)   ((e)->cipher->iv_len)
#define EVP_CIPHER_CTX_get_app_data(e) ((e)->app_data)
#define EVP_CIPHER_CTX_set_app_data(e,d) ((e)->app_data=(char *)(d))
#define EVP_CIPHER_CTX_type(c)        EVP_CIPHER_type(EVP_CIPHER_CTX_cipher(c))
#define EVP_CIPHER_CTX_flags(e)               ((e)->cipher->flags)
#define EVP_CIPHER_CTX_mode(e)        ((e)->cipher->flags & EVP_CIPH_MODE)

int EVP_CIPHER_param_to_asn1(EVP_CIPHER_CTX *c, ASN1_TYPE *type);
int EVP_CIPHER_asn1_to_param(EVP_CIPHER_CTX *c, ASN1_TYPE *type);
```

DESCRIPTION

The EVP cipher routines are a high level interface to certain symmetric ciphers.

EVP_CIPHER_CTX_init() initializes cipher contex **ctx**.

EVP_EncryptInit_ex() sets up cipher context **ctx** for encryption with cipher **type** from ENGINE **impl**. **ctx** must be initialized before calling this function. **type** is normally supplied by a function such as *EVP_des_cbc()*. If **impl** is NULL then the default implementation is used. **key** is the symmetric key to use and **iv** is the IV to use (if necessary), the actual number of bytes used for the key and IV depends on the cipher. It is possible to set all parameters to NULL except **type** in an initial call and supply the remaining parameters in subsequent calls, all of which have **type** set to NULL. This is done when the default cipher parameters are not appropriate.

EVP_EncryptUpdate() encrypts **inl** bytes from the buffer **in** and writes the encrypted version to **out**. This function can be called multiple times to encrypt successive blocks of data. The amount of data written depends on the block alignment of the encrypted data: as a result the amount of data written may be anything from zero bytes to (inl + cipher_block_size − 1) so **outl** should contain sufficient room. The actual number of bytes written is placed in **outl**.

If padding is enabled (the default) then *EVP_EncryptFinal_ex()* encrypts the ''final'' data, that is any data that remains in a partial block. It uses standard block padding (aka PKCS padding). The encrypted final data is written to **out** which should have sufficient space for one cipher block. The number of bytes written is placed in **outl**. After this function is called the encryption operation is finished and no further calls to *EVP_EncryptUpdate()* should be made.

If padding is disabled then *EVP_EncryptFinal_ex()* will not encrypt any more data and it will return an error if any data remains in a partial block: that is if the total data length is not a multiple of the block size.

EVP_DecryptInit_ex(), *EVP_DecryptUpdate()* and *EVP_DecryptFinal_ex()* are the corresponding decryption operations. *EVP_DecryptFinal()* will return an error code if padding is enabled and the final block is

not correctly formatted. The parameters and restrictions are identical to the encryption operations except that if padding is enabled the decrypted data buffer **out** passed to *EVP_DecryptUpdate()* should have sufficient room for (**inl** + cipher_block_size) bytes unless the cipher block size is 1 in which case **inl** bytes is sufficient.

EVP_CipherInit_ex(), EVP_CipherUpdate() and *EVP_CipherFinal_ex()* are functions that can be used for decryption or encryption. The operation performed depends on the value of the **enc** parameter. It should be set to 1 for encryption, 0 for decryption and −1 to leave the value unchanged (the actual value of 'enc' being supplied in a previous call).

EVP_CIPHER_CTX_cleanup() clears all information from a cipher context and free up any allocated memory associate with it. It should be called after all operations using a cipher are complete so sensitive information does not remain in memory.

EVP_EncryptInit(), EVP_DecryptInit() and *EVP_CipherInit()* behave in a similar way to *EVP_EncryptInit_ex()*, EVP_DecryptInit_ex and *EVP_CipherInit_ex()* except the **ctx** paramter does not need to be initialized and they always use the default cipher implementation.

EVP_EncryptFinal(), EVP_DecryptFinal() and *EVP_CipherFinal()* behave in a similar way to *EVP_EncryptFinal_ex(), EVP_DecryptFinal_ex()* and *EVP_CipherFinal_ex()* except **ctx** is automatically cleaned up after the call.

EVP_get_cipherbyname(), EVP_get_cipherbynid() and *EVP_get_cipherbyobj()* return an EVP_CIPHER structure when passed a cipher name, a NID or an ASN1_OBJECT structure.

EVP_CIPHER_nid() and *EVP_CIPHER_CTX_nid()* return the NID of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The actual NID value is an internal value which may not have a corresponding OBJECT IDENTIFIER.

EVP_CIPHER_CTX_set_padding() enables or disables padding. By default encryption operations are padded using standard block padding and the padding is checked and removed when decrypting. If the **pad** parameter is zero then no padding is performed, the total amount of data encrypted or decrypted must then be a multiple of the block size or an error will occur.

EVP_CIPHER_key_length() and *EVP_CIPHER_CTX_key_length()* return the key length of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The constant **EVP_MAX_KEY_LENGTH** is the maximum key length for all ciphers. Note: although *EVP_CIPHER_key_length()* is fixed for a given cipher, the value of *EVP_CIPHER_CTX_key_length()* may be different for variable key length ciphers.

EVP_CIPHER_CTX_set_key_length() sets the key length of the cipher ctx. If the cipher is a fixed length cipher then attempting to set the key length to any value other than the fixed value is an error.

EVP_CIPHER_iv_length() and *EVP_CIPHER_CTX_iv_length()* return the IV length of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX**. It will return zero if the cipher does not use an IV. The constant **EVP_MAX_IV_LENGTH** is the maximum IV length for all ciphers.

EVP_CIPHER_block_size() and *EVP_CIPHER_CTX_block_size()* return the block size of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The constant **EVP_MAX_IV_LENGTH** is also the maximum block length for all ciphers.

EVP_CIPHER_type() and *EVP_CIPHER_CTX_type()* return the type of the passed cipher or context. This ''type'' is the actual NID of the cipher OBJECT IDENTIFIER as such it ignores the cipher parameters and 40 bit RC2 and 128 bit RC2 have the same NID. If the cipher does not have an object identifier or does not have ASN1 support this function will return **NID_undef**.

EVP_CIPHER_CTX_cipher() returns the **EVP_CIPHER** structure when passed an **EVP_CIPHER_CTX** structure.

EVP_CIPHER_mode() and *EVP_CIPHER_CTX_mode()* return the block cipher mode: EVP_CIPH_ECB_MODE, EVP_CIPH_CBC_MODE, EVP_CIPH_CFB_MODE or EVP_CIPH_OFB_MODE. If the cipher is a stream cipher then EVP_CIPH_STREAM_CIPHER is returned.

EVP_CIPHER_param_to_asn1() sets the AlgorithmIdentifier ''parameter'' based on the passed cipher. This

will typically include any parameters and an IV. The cipher IV (if any) must be set when this call is made. This call should be made before the cipher is actually ''used'' (before any *EVP_EncryptUpdate()*, *EVP_DecryptUpdate()* calls for example). This function may fail if the cipher does not have any ASN1 support.

EVP_CIPHER_asn1_to_param() sets the cipher parameters based on an ASN1 AlgorithmIdentifier ''parameter''. The precise effect depends on the cipher In the case of RC2, for example, it will set the IV and effective key length. This function should be called after the base cipher type is set but before the key is set. For example *EVP_CipherInit()* will be called with the IV and key set to NULL, *EVP_CIPHER_asn1_to_param()* will be called and finally *EVP_CipherInit()* again with all parameters except the key set to NULL. It is possible for this function to fail if the cipher does not have any ASN1 support or the parameters cannot be set (for example the RC2 effective key length is not supported.

EVP_CIPHER_CTX_ctrl() allows various cipher specific parameters to be determined and set. Currently only the RC2 effective key length and the number of rounds of RC5 can be set.

RETURN VALUES

EVP_EncryptInit_ex(), *EVP_EncryptUpdate()* and *EVP_EncryptFinal_ex()* return 1 for success and 0 for failure.

EVP_DecryptInit_ex() and *EVP_DecryptUpdate()* return 1 for success and 0 for failure. *EVP_DecryptFinal_ex()* returns 0 if the decrypt failed or 1 for success.

EVP_CipherInit_ex() and *EVP_CipherUpdate()* return 1 for success and 0 for failure. *EVP_CipherFinal_ex()* returns 0 for a decryption failure or 1 for success.

EVP_CIPHER_CTX_cleanup() returns 1 for success and 0 for failure.

EVP_get_cipherbyname(), *EVP_get_cipherbynid()* and *EVP_get_cipherbyobj()* return an **EVP_CIPHER** structure or NULL on error.

EVP_CIPHER_nid() and *EVP_CIPHER_CTX_nid()* return a NID.

EVP_CIPHER_block_size() and *EVP_CIPHER_CTX_block_size()* return the block size.

EVP_CIPHER_key_length() and *EVP_CIPHER_CTX_key_length()* return the key length.

EVP_CIPHER_CTX_set_padding() always returns 1.

EVP_CIPHER_iv_length() and *EVP_CIPHER_CTX_iv_length()* return the IV length or zero if the cipher does not use an IV.

EVP_CIPHER_type() and *EVP_CIPHER_CTX_type()* return the NID of the cipher's OBJECT IDENTIFIER or NID_undef if it has no defined OBJECT IDENTIFIER.

EVP_CIPHER_CTX_cipher() returns an **EVP_CIPHER** structure.

EVP_CIPHER_param_to_asn1() and *EVP_CIPHER_asn1_to_param()* return 1 for success or zero for failure.

CIPHER LISTING

All algorithms have a fixed key length unless otherwise stated.

EVP_enc_null()
: Null cipher: does nothing.

EVP_des_cbc(void), EVP_des_ecb(void), EVP_des_cfb(void), EVP_des_ofb(void)
: DES in CBC, ECB, CFB and OFB modes respectively.

EVP_des_ede_cbc(void), *EVP_des_ede()*, EVP_des_ede_ofb(void), EVP_des_ede_cfb(void)
: Two key triple DES in CBC, ECB, CFB and OFB modes respectively.

EVP_des_ede3_cbc(void), *EVP_des_ede3()*, EVP_des_ede3_ofb(void), EVP_des_ede3_cfb(void)
: Three key triple DES in CBC, ECB, CFB and OFB modes respectively.

EVP_desx_cbc(void)
DESX algorithm in CBC mode.

EVP_rc4(void)
RC4 stream cipher. This is a variable key length cipher with default key length 128 bits.

EVP_rc4_40(void)
RC4 stream cipher with 40 bit key length. This is obsolete and new code should use *EVP_rc4()* and the *EVP_CIPHER_CTX_set_key_length()* function.

EVP_idea_cbc() EVP_idea_ecb(void), EVP_idea_cfb(void), EVP_idea_ofb(void), EVP_idea_cbc(void)
IDEA encryption algorithm in CBC, ECB, CFB and OFB modes respectively.

EVP_rc2_cbc(void), EVP_rc2_ecb(void), EVP_rc2_cfb(void), EVP_rc2_ofb(void)
RC2 encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher with an additional parameter called ''effective key bits'' or ''effective key length''. By default both are set to 128 bits.

EVP_rc2_40_cbc(void), EVP_rc2_64_cbc(void)
RC2 algorithm in CBC mode with a default key length and effective key length of 40 and 64 bits. These are obsolete and new code should use *EVP_rc2_cbc()*, *EVP_CIPHER_CTX_set_key_length()* and *EVP_CIPHER_CTX_ctrl()* to set the key length and effective key length.

EVP_bf_cbc(void), EVP_bf_ecb(void), EVP_bf_cfb(void), EVP_bf_ofb(void);
Blowfish encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher.

EVP_cast5_cbc(void), EVP_cast5_ecb(void), EVP_cast5_cfb(void), EVP_cast5_ofb(void)
CAST encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher.

EVP_rc5_32_12_16_cbc(void), EVP_rc5_32_12_16_ecb(void), EVP_rc5_32_12_16_cfb(void), EVP_rc5_32_12_16_ofb(void)
RC5 encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher with an additional ''number of rounds'' parameter. By default the key length is set to 128 bits and 12 rounds.

NOTES

Where possible the **EVP** interface to symmetric ciphers should be used in preference to the low level interfaces. This is because the code then becomes transparent to the cipher used and much more flexible.

PKCS padding works by adding **n** padding bytes of value **n** to make the total length of the encrypted data a multiple of the block size. Padding is always added so if the data is already a multiple of the block size **n** will equal the block size. For example if the block size is 8 and 11 bytes are to be encrypted then 5 padding bytes of value 5 will be added.

When decrypting the final block is checked to see if it has the correct form.

Although the decryption operation can produce an error if padding is enabled, it is not a strong test that the input data or key is correct. A random block has better than 1 in 256 chance of being of the correct format and problems with the input data earlier on will not produce a final decrypt error.

If padding is disabled then the decryption operation will always succeed if the total amount of data decrypted is a multiple of the block size.

The functions *EVP_EncryptInit()*, *EVP_EncryptFinal()*, *EVP_DecryptInit()*, *EVP_CipherInit()* and *EVP_CipherFinal()* are obsolete but are retained for compatibility with existing code. New code should use *EVP_EncryptInit_ex()*, *EVP_EncryptFinal_ex()*, *EVP_DecryptInit_ex()*, *EVP_DecryptFinal_ex()*, *EVP_CipherInit_ex()* and *EVP_CipherFinal_ex()* because they can reuse an existing context without allocating and freeing it up on each call.

BUGS

For RC5 the number of rounds can currently only be set to 8, 12 or 16. This is a limitation of the current RC5 code rather than the EVP interface.

EVP_MAX_KEY_LENGTH and EVP_MAX_IV_LENGTH only refer to the internal ciphers with default key lengths. If custom ciphers exceed these values the results are unpredictable. This is because it has become standard practice to define a generic key as a fixed unsigned char array containing EVP_MAX_KEY_LENGTH bytes.

The ASN1 code is incomplete (and sometimes inaccurate) it has only been tested for certain common S/MIME ciphers (RC2, DES, triple DES) in CBC mode.

EXAMPLES

Get the number of rounds used in RC5:

```
int nrounds;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GET_RC5_ROUNDS, 0, &nrounds);
```

Get the RC2 effective key length:

```
int key_bits;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GET_RC2_KEY_BITS, 0, &key_bits);
```

Set the number of rounds used in RC5:

```
int nrounds;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_SET_RC5_ROUNDS, nrounds, NULL);
```

Set the effective key length used in RC2:

```
int key_bits;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_SET_RC2_KEY_BITS, key_bits, NULL);
```

Encrypt a string using blowfish:

```
int do_crypt(char *outfile)
       {
       unsigned char outbuf[1024];
       int outlen, tmplen;
       /* Bogus key and IV: we'd normally set these from
        * another source.
        */
       unsigned char key[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
       unsigned char iv[] = {1,2,3,4,5,6,7,8};
       char intext[] = "Some Crypto Text";
       EVP_CIPHER_CTX ctx;
       FILE *out;
       EVP_CIPHER_CTX_init(&ctx);
       EVP_EncryptInit_ex(&ctx, EVP_bf_cbc(), NULL, key, iv);
```

```
if(!EVP_EncryptUpdate(&ctx, outbuf, &outlen, intext, strlen(intext)))
        {
        /* Error */
        return 0;
        }
/* Buffer passed to EVP_EncryptFinal() must be after data just
 * encrypted to avoid overwriting it.
 */
if(!EVP_EncryptFinal_ex(&ctx, outbuf + outlen, &tmplen))
        {
        /* Error */
        return 0;
        }
outlen += tmplen;
EVP_CIPHER_CTX_cleanup(&ctx);
/* Need binary mode for fopen because encrypted data is
 * binary data. Also cannot use strlen() on it because
 * it wont be null terminated and may contain embedded
 * nulls.
 */
out = fopen(outfile, "wb");
fwrite(outbuf, 1, outlen, out);
fclose(out);
return 1;
}
```

The ciphertext from the above example can be decrypted using the **openssl** utility with the command line:

```
S<openssl bf -in cipher.bin -K 000102030405060708090A0B0C0D0E0F -iv 010203040506070
```

General encryption, decryption function example using FILE I/O and RC2 with an 80 bit key:

```
int do_crypt(FILE *in, FILE *out, int do_encrypt)
        {
        /* Allow enough space in output buffer for additional block */
        inbuf[1024], outbuf[1024 + EVP_MAX_BLOCK_LENGTH];
        int inlen, outlen;
        /* Bogus key and IV: we'd normally set these from
         * another source.
         */
        unsigned char key[] = "0123456789";
        unsigned char iv[] = "12345678";
        /* Don't set key or IV because we will modify the parameters */
        EVP_CIPHER_CTX_init(&ctx);
        EVP_CipherInit_ex(&ctx, EVP_rc2(), NULL, NULL, NULL, do_encrypt);
        EVP_CIPHER_CTX_set_key_length(&ctx, 10);
        /* We finished modifying parameters so now we can set key and IV */
        EVP_CipherInit_ex(&ctx, NULL, NULL, key, iv, do_encrypt);
```

```
for(;;)
        {
        inlen = fread(inbuf, 1, 1024, in);
        if(inlen <= 0) break;
        if(!EVP_CipherUpdate(&ctx, outbuf, &outlen, inbuf, inlen))
                {
                /* Error */
                EVP_CIPHER_CTX_cleanup(&ctx);
                return 0;
                }
        fwrite(outbuf, 1, outlen, out);
        }
if(!EVP_CipherFinal_ex(&ctx, outbuf, &outlen))
        {
        /* Error */
        EVP_CIPHER_CTX_cleanup(&ctx);
        return 0;
        }
fwrite(outbuf, 1, outlen, out);

EVP_CIPHER_CTX_cleanup(&ctx);
return 1;
}
```

SEE ALSO

openssl_evp (3)

HISTORY

EVP_CIPHER_CTX_init(), *EVP_EncryptInit_ex()*, *EVP_EncryptFinal_ex()*, *EVP_DecryptInit_ex()*, *EVP_DecryptFinal_ex()*, *EVP_CipherInit_ex()*, *EVP_CipherFinal_ex()* and *EVP_CIPHER_CTX_set_padding()* appeared in OpenSSL 0.9.7.

NAME

EVP_OpenInit, EVP_OpenUpdate, EVP_OpenFinal − EVP envelope decryption

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>

int EVP_OpenInit(EVP_CIPHER_CTX *ctx,EVP_CIPHER *type,unsigned char *ek,
                 int ekl,unsigned char *iv,EVP_PKEY *priv);
int EVP_OpenUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl, unsigned char *in, int inl);
int EVP_OpenFinal(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl);
```

DESCRIPTION

The EVP envelope routines are a high level interface to envelope decryption. They decrypt a public key encrypted symmetric key and then decrypt data using it.

EVP_OpenInit() initializes a cipher context **ctx** for decryption with cipher **type**. It decrypts the encrypted symmetric key of length **ekl** bytes passed in the **ek** parameter using the private key **priv**. The IV is supplied in the **iv** parameter.

EVP_OpenUpdate() and *EVP_OpenFinal()* have exactly the same properties as the *EVP_DecryptUpdate()* and *EVP_DecryptFinal()* routines, as documented on the *EVP_EncryptInit* (3) manual page.

NOTES

It is possible to call *EVP_OpenInit()* twice in the same way as *EVP_DecryptInit()*. The first call should have **priv** set to NULL and (after setting any cipher parameters) it should be called again with **type** set to NULL.

If the cipher passed in the **type** parameter is a variable length cipher then the key length will be set to the value of the recovered key length. If the cipher is a fixed length cipher then the recovered key length must match the fixed cipher length.

RETURN VALUES

EVP_OpenInit() returns 0 on error or a non zero integer (actually the recovered secret key size) if successful.

EVP_OpenUpdate() returns 1 for success or 0 for failure.

EVP_OpenFinal() returns 0 if the decrypt failed or 1 for success.

SEE ALSO

openssl_evp (3), *openssl_rand* (3), *EVP_EncryptInit* (3), *EVP_SealInit* (3)

HISTORY

NAME

EVP_PKEY_new, EVP_PKEY_free − private key allocation functions.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>

EVP_PKEY *EVP_PKEY_new(void);
void EVP_PKEY_free(EVP_PKEY *key);
```

DESCRIPTION

The *EVP_PKEY_new()* function allocates an empty **EVP_PKEY** structure which is used by OpenSSL to store private keys.

EVP_PKEY_free() frees up the private key **key**.

NOTES

The **EVP_PKEY** structure is used by various OpenSSL functions which require a general private key without reference to any particular algorithm.

The structure returned by *EVP_PKEY_new()* is empty. To add a private key to this empty structure the functions described in *EVP_PKEY_set1_RSA* (3) should be used.

RETURN VALUES

EVP_PKEY_new() returns either the newly allocated **EVP_PKEY** structure of **NULL** if an error occurred.

EVP_PKEY_free() does not return a value.

SEE ALSO

EVP_PKEY_set1_RSA (3)

HISTORY

TBA

NAME

EVP_PKEY_set1_RSA, EVP_PKEY_set1_DSA, EVP_PKEY_set1_DH, EVP_PKEY_set1_EC_KEY, EVP_PKEY_get1_RSA, EVP_PKEY_get1_DSA, EVP_PKEY_get1_DH, EVP_PKEY_get1_EC_KEY, EVP_PKEY_assign_RSA, EVP_PKEY_assign_DSA, EVP_PKEY_assign_DH, EVP_PKEY_assign_EC_KEY, EVP_PKEY_type − EVP_PKEY assignment functions.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>

int EVP_PKEY_set1_RSA(EVP_PKEY *pkey,RSA *key);
int EVP_PKEY_set1_DSA(EVP_PKEY *pkey,DSA *key);
int EVP_PKEY_set1_DH(EVP_PKEY *pkey,DH *key);
int EVP_PKEY_set1_EC_KEY(EVP_PKEY *pkey,EC_KEY *key);

RSA *EVP_PKEY_get1_RSA(EVP_PKEY *pkey);
DSA *EVP_PKEY_get1_DSA(EVP_PKEY *pkey);
DH *EVP_PKEY_get1_DH(EVP_PKEY *pkey);
EC_KEY *EVP_PKEY_get1_EC_KEY(EVP_PKEY *pkey);

int EVP_PKEY_assign_RSA(EVP_PKEY *pkey,RSA *key);
int EVP_PKEY_assign_DSA(EVP_PKEY *pkey,DSA *key);
int EVP_PKEY_assign_DH(EVP_PKEY *pkey,DH *key);
int EVP_PKEY_assign_EC_KEY(EVP_PKEY *pkey,EC_KEY *key);

int EVP_PKEY_type(int type);
```

DESCRIPTION

EVP_PKEY_set1_RSA(), *EVP_PKEY_set1_DSA()*, *EVP_PKEY_set1_DH()* and *EVP_PKEY_set1_EC_KEY()* set the key referenced by **pkey** to **key**.

EVP_PKEY_get1_RSA(), *EVP_PKEY_get1_DSA()*, *EVP_PKEY_get1_DH()* and *EVP_PKEY_get1_EC_KEY()* return the referenced key in **pkey** or **NULL** if the key is not of the correct type.

EVP_PKEY_assign_RSA() *EVP_PKEY_assign_DSA()*, *EVP_PKEY_assign_DH()* and *EVP_PKEY_assign_EC_KEY()* also set the referenced key to **key** however these use the supplied **key** internally and so **key** will be freed when the parent **pkey** is freed.

EVP_PKEY_type() returns the type of key corresponding to the value **type**. The type of a key can be obtained with EVP_PKEY_type(pkey−>type). The return value will be EVP_PKEY_RSA, EVP_PKEY_DSA, EVP_PKEY_DH or EVP_PKEY_EC for the corresponding key types or NID_undef if the key type is unassigned.

NOTES

In accordance with the OpenSSL naming convention the key obtained from or assigned to the **pkey** using the **1** functions must be freed as well as **pkey**.

EVP_PKEY_assign_RSA() *EVP_PKEY_assign_DSA()*, *EVP_PKEY_assign_DH()* *EVP_PKEY_assign_EC_KEY()* are implemented as macros.

RETURN VALUES

EVP_PKEY_set1_RSA(), *EVP_PKEY_set1_DSA()*, *EVP_PKEY_set1_DH()* and *EVP_PKEY_set1_EC_KEY()* return 1 for success or 0 for failure.

EVP_PKEY_get1_RSA(), *EVP_PKEY_get1_DSA()*, *EVP_PKEY_get1_DH()* and *EVP_PKEY_get1_EC_KEY()* return the referenced key or **NULL** if an error occurred.

EVP_PKEY_assign_RSA() *EVP_PKEY_assign_DSA()*, *EVP_PKEY_assign_DH()* and *EVP_PKEY_assign_EC_KEY()* return 1 for success and 0 for failure.

SEE ALSO

EVP_PKEY_new (3)

HISTORY

TBA

NAME

EVP_SealInit, EVP_SealUpdate, EVP_SealFinal − EVP envelope encryption

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>

int EVP_SealInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                 unsigned char **ek, int *ekl, unsigned char *iv,
                 EVP_PKEY **pubk, int npubk);
int EVP_SealUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl, unsigned char *in, int inl);
int EVP_SealFinal(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl);
```

DESCRIPTION

The EVP envelope routines are a high level interface to envelope encryption. They generate a random key and IV (if required) then ''envelope'' it by using public key encryption. Data can then be encrypted using this key.

EVP_SealInit() initializes a cipher context **ctx** for encryption with cipher **type** using a random secret key and IV. **type** is normally supplied by a function such as *EVP_des_cbc()*. The secret key is encrypted using one or more public keys, this allows the same encrypted data to be decrypted using any of the corresponding private keys. **ek** is an array of buffers where the public key encrypted secret key will be written, each buffer must contain enough room for the corresponding encrypted key: that is **ek[i]** must have room for **EVP_PKEY_size(pubk[i])** bytes. The actual size of each encrypted secret key is written to the array **ekl**. **pubk** is an array of **npubk** public keys.

The **iv** parameter is a buffer where the generated IV is written to. It must contain enough room for the corresponding cipher's IV, as determined by (for example) EVP_CIPHER_iv_length(type).

If the cipher does not require an IV then the **iv** parameter is ignored and can be **NULL**.

EVP_SealUpdate() and *EVP_SealFinal()* have exactly the same properties as the *EVP_EncryptUpdate()* and *EVP_EncryptFinal()* routines, as documented on the *EVP_EncryptInit* (3) manual page.

RETURN VALUES

EVP_SealInit() returns 0 on error or **npubk** if successful.

EVP_SealUpdate() and *EVP_SealFinal()* return 1 for success and 0 for failure.

NOTES

Because a random secret key is generated the random number generator must be seeded before calling *EVP_SealInit()*.

The public key must be RSA because it is the only OpenSSL public key algorithm that supports key transport.

Envelope encryption is the usual method of using public key encryption on large amounts of data, this is because public key encryption is slow but symmetric encryption is fast. So symmetric encryption is used for bulk encryption and the small random symmetric key used is transferred using public key encryption.

It is possible to call *EVP_SealInit()* twice in the same way as *EVP_EncryptInit()*. The first call should have **npubk** set to 0 and (after setting any cipher parameters) it should be called again with **type** set to NULL.

SEE ALSO

openssl_evp (3), *openssl_rand* (3), *EVP_EncryptInit* (3), *EVP_OpenInit* (3)

HISTORY

EVP_SealFinal() did not return a value before OpenSSL 0.9.7.

NAME

EVP_SignInit, EVP_SignUpdate, EVP_SignFinal − EVP signing functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>

int EVP_SignInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int EVP_SignUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
int EVP_SignFinal(EVP_MD_CTX *ctx,unsigned char *sig,unsigned int *s, EVP_PKEY *pke

void EVP_SignInit(EVP_MD_CTX *ctx, const EVP_MD *type);

int EVP_PKEY_size(EVP_PKEY *pkey);
```

DESCRIPTION

The EVP signature routines are a high level interface to digital signatures.

EVP_SignInit_ex() sets up signing context **ctx** to use digest **type** from ENGINE **impl**. **ctx** must be initialized with *EVP_MD_CTX_init()* before calling this function.

EVP_SignUpdate() hashes **cnt** bytes of data at **d** into the signature context **ctx**. This function can be called several times on the same **ctx** to include additional data.

EVP_SignFinal() signs the data in **ctx** using the private key **pkey** and places the signature in **sig**. The number of bytes of data written (i.e. the length of the signature) will be written to the integer at **s**, at most EVP_PKEY_size(pkey) bytes will be written.

EVP_SignInit() initializes a signing context **ctx** to use the default implementation of digest **type**.

EVP_PKEY_size() returns the maximum size of a signature in bytes. The actual signature returned by *EVP_SignFinal()* may be smaller.

RETURN VALUES

EVP_SignInit_ex(), *EVP_SignUpdate()* and *EVP_SignFinal()* return 1 for success and 0 for failure.

EVP_PKEY_size() returns the maximum size of a signature in bytes.

The error codes can be obtained by *ERR_get_error* (3).

NOTES

The **EVP** interface to digital signatures should almost always be used in preference to the low level interfaces. This is because the code then becomes transparent to the algorithm used and much more flexible.

Due to the link between message digests and public key algorithms the correct digest algorithm must be used with the correct public key type. A list of algorithms and associated public key algorithms appears in *EVP_DigestInit* (3).

When signing with DSA private keys the random number generator must be seeded or the operation will fail. The random number generator does not need to be seeded for RSA signatures.

The call to *EVP_SignFinal()* internally finalizes a copy of the digest context. This means that calls to *EVP_SignUpdate()* and *EVP_SignFinal()* can be called later to digest and sign additional data.

Since only a copy of the digest context is ever finalized the context must be cleaned up after use by calling *EVP_MD_CTX_cleanup()* or a memory leak will occur.

BUGS

Older versions of this documentation wrongly stated that calls to *EVP_SignUpdate()* could not be made after calling *EVP_SignFinal()*.

Since the private key is passed in the call to *EVP_SignFinal()* any error relating to the private key (for example an unsuitable key and digest combination) will not be indicated until after potentially large amounts of data have been passed through *EVP_SignUpdate()*.

It is not possible to change the signing parameters using these function.

The previous two bugs are fixed in the newer EVP_SignDigest*() function.

SEE ALSO

EVP_VerifyInit (3), *EVP_DigestInit* (3), *openssl_err* (3), *openssl_evp* (3), *openssl_hmac* (3), *md2* (3), *openssl_md5* (3), *openssl_mdc2* (3), *openssl_ripemd* (3), *openssl_sha* (3), *openssl_dgst* (1)

HISTORY

EVP_SignInit(), *EVP_SignUpdate()* and *EVP_SignFinal()* are available in all versions of SSLeay and OpenSSL.

EVP_SignInit_ex() was added in OpenSSL 0.9.7.

NAME

EVP_VerifyInit, EVP_VerifyUpdate, EVP_VerifyFinal − EVP signature verification functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>

int EVP_VerifyInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int EVP_VerifyUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
int EVP_VerifyFinal(EVP_MD_CTX *ctx,unsigned char *sigbuf, unsigned int siglen,EVP_

int EVP_VerifyInit(EVP_MD_CTX *ctx, const EVP_MD *type);
```

DESCRIPTION

The EVP signature verification routines are a high level interface to digital signatures.

EVP_VerifyInit_ex() sets up verification context **ctx** to use digest **type** from ENGINE **impl**. **ctx** must be initialized by calling *EVP_MD_CTX_init()* before calling this function.

EVP_VerifyUpdate() hashes **cnt** bytes of data at **d** into the verification context **ctx**. This function can be called several times on the same **ctx** to include additional data.

EVP_VerifyFinal() verifies the data in **ctx** using the public key **pkey** and against the **siglen** bytes at **sigbuf**.

EVP_VerifyInit() initializes verification context **ctx** to use the default implementation of digest **type**.

RETURN VALUES

EVP_VerifyInit_ex() and *EVP_VerifyUpdate()* return 1 for success and 0 for failure.

EVP_VerifyFinal() returns 1 for a correct signature, 0 for failure and −1 if some other error occurred.

The error codes can be obtained by *ERR_get_error* (3).

NOTES

The **EVP** interface to digital signatures should almost always be used in preference to the low level interfaces. This is because the code then becomes transparent to the algorithm used and much more flexible.

Due to the link between message digests and public key algorithms the correct digest algorithm must be used with the correct public key type. A list of algorithms and associated public key algorithms appears in *EVP_DigestInit* (3).

The call to *EVP_VerifyFinal()* internally finalizes a copy of the digest context. This means that calls to *EVP_VerifyUpdate()* and *EVP_VerifyFinal()* can be called later to digest and verify additional data.

Since only a copy of the digest context is ever finalized the context must be cleaned up after use by calling *EVP_MD_CTX_cleanup()* or a memory leak will occur.

BUGS

Older versions of this documentation wrongly stated that calls to *EVP_VerifyUpdate()* could not be made after calling *EVP_VerifyFinal()*.

Since the public key is passed in the call to *EVP_SignFinal()* any error relating to the private key (for example an unsuitable key and digest combination) will not be indicated until after potentially large amounts of data have been passed through *EVP_SignUpdate()*.

It is not possible to change the signing parameters using these function.

The previous two bugs are fixed in the newer EVP_VerifyDigest*() function.

SEE ALSO

openssl_evp (3), *EVP_SignInit* (3), *EVP_DigestInit* (3), *openssl_err* (3), *openssl_evp* (3), *openssl_hmac* (3), *md2* (3), *openssl_md5* (3), *openssl_mdc2* (3), *openssl_ripemd* (3), *openssl_sha* (3), *openssl_dgst* (1)

HISTORY

EVP_VerifyInit(), *EVP_VerifyUpdate()* and *EVP_VerifyFinal()* are available in all versions of SSLeay and OpenSSL.

EVP_VerifyInit_ex() was added in OpenSSL 0.9.7

NAME

C++Intro − Introduction to the GNU libstdc++-v3 man pages

DESCRIPTION

This man page serves as a brief introduction to the GNU implementation of the Standard C++ Library. For a better introduction and more complete documentation, see the **libstdc++-v3** homepage listed at the end.

All standard library entities are declared within *namespace std* and have manual entries beginning with "std::". For example, to see documentation of the template class *std::vector* one would use "man std::vector". Some entities do not have a separate man page; for those see the main listing in "man Namespace_std".

All the man pages are automatically generated by Doxygen. For more information on this tool, see the HTML counterpart to these man pages.

Some man pages do not correspond to individual classes or functions. Rather they describe categories of the Standard Library. (For a more thorough introduction to the various categories, consult a text such as Josuttis' or Austern's.) These category pages are:

C++Intro	This page.
Namespace_std	A listing of the contents of std::.
Namespace___gnu_cxx	A listing of the contents of __gnu_cxx::.
Containers	An introduction to container classes.
Sequences	Linear containers.
Assoc_containers	Key-based containers.
Iterator_types	Programatically distinguishing iterators/pointers.
Intro_functors	An introduction to function objects, or functors.
Arithmetic_functors	Functors for basic math.
Binder_functors	Functors which "remember" an argument.
Comparison_functors	Functors wrapping built-in comparisons.
Func_ptr_functors	Functors for use with pointers to functions.
Logical_functors	Functors wrapping the Boolean operations.
Member_ptr_functor	Functors for use with pointers to members.
Negation_functors	Functors which negate their contents.
SGIextensions	A list of the extensions from the SGI STL subset.

The HTML documentation typically goes into much more depth.

FILES

Lots!

Standard Headers

These headers will be found automatically, unless you instruct the compiler otherwise.

<algorithm> <csignal> <iomanip> <ostream>
<bitset> <cstdarg> <ios> <queue>
<cassert> <cstddef> <iosfwd> <set>
<cctype> <cstdio> <iostream> <sstream>
<cerrno> <cstdlib> <istream> <stack>
<cfloat> <cstring> <iterator> <stdexcept>
<ciso>646 <ctime> <limits> <streambuf>
<climits> <cwchar> <list> <string>
<clocale> <cwctype> <locale> <utility>
<cmath> <deque> <map> <valarray>
<complex> <fstream> <memory> <vector>
<csetjmp> <functional> <numeric>

Backwards-Compatibility Headers

For GCC 3.0 these headers will be found automatically, unless you instruct the compiler otherwise. You should not depend on this, instead you should read FAQ 5.4 and use a **backward/** prefix.

<algo.h> <hash_map.h> <map.h> <slist.h>
<algobase.h> <hash_set.h> <multimap.h> <stack.h>
<alloc.h> <hashtable.h> <multiset.h> <stream.h>
<bvector.h> <heap.h> <new.h> <streambuf.h>
<complex.h> <iomanip.h> <ostream.h> <strstream>
<defalloc.h> <iostream.h> <pair.h> <strstream.h>
<deque.h> <istream.h> <queue.h> <tempbuf.h>
<fstream.h> <iterator.h> <rope.h> <tree.h>
<function.h> <list.h> <set.h> <vector.h>

Extension Headers

These headers will only be found automatically if you include the leading **ext/** in the name. Otherwise you need to read FAQ 5.4.

<ext/algorithm> <ext/numeric>
<ext/functional> <ext/iterator>
<ext/slist> <ext/rb_tree>
<ext/hash_map> <ext/hash_set>
<ext/rope> <ext/memory>
<ext/bitmap_allocator.h> <ext/debug_allocator.h>
<ext/malloc_allocator.h> <ext/mt_allocator.h>
<ext/pool_allocator.h> <ext/pod_char_traits.h>
<ext/stdio_filebuf.h> <ext/stdio_sync_filebuf.h>

Libraries

libstdc++.a

The library implementation in static archive form. If you did not configure libstdc++-v3 to use shared libraries, this will always be used. Otherwise it will only be used if the user requests it.

libsupc++.a

This library contains C++ language support routines. Usually you will never need to know about it, but it can be useful. See FAQ 2.5.

libstdc++.so[.N]

The library implementation in shared object form. This will be used in preference to the static archive form by default. N will be a number equal to or greater than 3. If N is in the 2.x series, then you are looking at the old libstdc++-v2 library, which we do not maintain.

libstdc++.la

libsupc++.la

These are Libtool library files, and should only be used when working with that tool.

CONFORMING TO

Almost conforming to **International Standard ISO/IEC 14882:1998(E),** *Programming Languages* --- *C++* (aka the C++ standard), in addition to corrections proposed by the Library Working Group, JTC1/SC22/WG21.

SEE ALSO

http://gcc.gnu.org/libstdc++/ for the Frequently Asked Questions, online documentation, and much, much more!

NAME

OBJ_nid2obj, OBJ_nid2ln, OBJ_nid2sn, OBJ_obj2nid, OBJ_txt2nid, OBJ_ln2nid, OBJ_sn2nid, OBJ_cmp, OBJ_dup, OBJ_txt2obj, OBJ_obj2txt, OBJ_create, OBJ_cleanup − ASN1 object utility functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/objects.h>

ASN1_OBJECT * OBJ_nid2obj(int n);
const char *  OBJ_nid2ln(int n);
const char *  OBJ_nid2sn(int n);

int OBJ_obj2nid(const ASN1_OBJECT *o);
int OBJ_ln2nid(const char *ln);
int OBJ_sn2nid(const char *sn);

int OBJ_txt2nid(const char *s);

ASN1_OBJECT * OBJ_txt2obj(const char *s, int no_name);
int OBJ_obj2txt(char *buf, int buf_len, const ASN1_OBJECT *a, int no_name);

int OBJ_cmp(const ASN1_OBJECT *a,const ASN1_OBJECT *b);
ASN1_OBJECT * OBJ_dup(const ASN1_OBJECT *o);

int OBJ_create(const char *oid,const char *sn,const char *ln);
void OBJ_cleanup(void);
```

DESCRIPTION

The ASN1 object utility functions process ASN1_OBJECT structures which are a representation of the ASN1 OBJECT IDENTIFIER (OID) type.

OBJ_nid2obj(), *OBJ_nid2ln()* and *OBJ_nid2sn()* convert the NID **n** to an ASN1_OBJECT structure, its long name and its short name respectively, or **NULL** is an error occurred.

OBJ_obj2nid(), *OBJ_ln2nid()*, *OBJ_sn2nid()* return the corresponding NID for the object **o**, the long name <ln> or the short name <sn> respectively or NID_undef if an error occurred.

OBJ_txt2nid() returns NID corresponding to text string <s>. **s** can be a long name, a short name or the numerical respresentation of an object.

OBJ_txt2obj() converts the text string **s** into an ASN1_OBJECT structure. If **no_name** is 0 then long names and short names will be interpreted as well as numerical forms. If **no_name** is 1 only the numerical form is acceptable.

OBJ_obj2txt() converts the **ASN1_OBJECT a** into a textual representation. The representation is written as a null terminated string to **buf** at most **buf_len** bytes are written, truncating the result if necessary. The total amount of space required is returned. If **no_name** is 0 then if the object has a long or short name then that will be used, otherwise the numerical form will be used. If **no_name** is 1 then the numerical form will always be used.

OBJ_cmp() compares **a** to **b**. If the two are identical 0 is returned.

OBJ_dup() returns a copy of **o**.

OBJ_create() adds a new object to the internal table. **oid** is the numerical form of the object, **sn** the short name and **ln** the long name. A new NID is returned for the created object.

OBJ_cleanup() cleans up OpenSSLs internal object table: this should be called before an application exits if any new objects were added using *OBJ_create()*.

NOTES

Objects in OpenSSL can have a short name, a long name and a numerical identifier (NID) associated with them. A standard set of objects is represented in an internal table. The appropriate values are defined in the header file **objects.h**.

For example the OID for commonName has the following definitions:

```
#define SN_commonName                   "CN"
#define LN_commonName                   "commonName"
#define NID_commonName                  13
```

New objects can be added by calling *OBJ_create()*.

Table objects have certain advantages over other objects: for example their NIDs can be used in a C language switch statement. They are also static constant structures which are shared: that is there is only a single constant structure for each table object.

Objects which are not in the table have the NID value NID_undef.

Objects do not need to be in the internal tables to be processed, the functions *OBJ_txt2obj()* and *OBJ_obj2txt()* can process the numerical form of an OID.

EXAMPLES

Create an object for **commonName**:

```
ASN1_OBJECT *o;
o = OBJ_nid2obj(NID_commonName);
```

Check if an object is **commonName**

```
if (OBJ_obj2nid(obj) == NID_commonName)
       /* Do something */
```

Create a new NID and initialize an object from it:

```
int new_nid;
ASN1_OBJECT *obj;
new_nid = OBJ_create("1.2.3.4", "NewOID", "New Object Identifier");
obj = OBJ_nid2obj(new_nid);
```

Create a new object directly:

```
obj = OBJ_txt2obj("1.2.3.4", 1);
```

BUGS

OBJ_obj2txt() is awkward and messy to use: it doesn't follow the convention of other OpenSSL functions where the buffer can be set to **NULL** to determine the amount of data that should be written. Instead **buf** must point to a valid buffer and **buf_len** should be set to a positive value. A buffer length of 80 should be more than enough to handle any OID encountered in practice.

RETURN VALUES

OBJ_nid2obj() returns an **ASN1_OBJECT** structure or **NULL** is an error occurred.

OBJ_nid2ln() and *OBJ_nid2sn()* returns a valid string or **NULL** on error.

OBJ_obj2nid(), *OBJ_ln2nid()*, *OBJ_sn2nid()* and *OBJ_txt2nid()* return a NID or **NID_undef** on error.

SEE ALSO

ERR_get_error (3)

HISTORY

TBA

NAME

OPENSSL_Applink − glue between OpenSSL BIO and Win32 compiler run−time

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
__declspec(dllexport) void **OPENSSL_Applink();
```

DESCRIPTION

OPENSSL_Applink is application-side interface which provides a glue between OpenSSL BIO layer and Win32 compiler run-time environment. Even though it appears at application side, it's essentially OpenSSL private interface. For this reason application developers are not expected to implement it, but to compile provided module with compiler of their choice and link it into the target application. The referred module is available as <openssl>/ms/applink.c.

NAME

OPENSSL_VERSION_NUMBER, SSLeay, SSLeay_version − get OpenSSL version number

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/opensslv.h>
#define OPENSSL_VERSION_NUMBER 0xnnnnnnnnnL

#include <openssl/crypto.h>
long SSLeay(void);
const char *SSLeay_version(int t);
```

DESCRIPTION

OPENSSL_VERSION_NUMBER is a numeric release version identifier:

```
MMNNFFPPS: major minor fix patch status
```

The status nibble has one of the values 0 for development, 1 to e for betas 1 to 14, and f for release.

for example

```
0x000906000 == 0.9.6 dev
0x000906023 == 0.9.6b beta 3
0x00090605f == 0.9.6e release
```

Versions prior to 0.9.3 have identifiers < 0x0930. Versions between 0.9.3 and 0.9.5 had a version identifier with this interpretation:

```
MMNNFFRBB major minor fix final beta/patch
```

for example

```
0x000904100 == 0.9.4 release
0x000905000 == 0.9.5 dev
```

Version 0.9.5a had an interim interpretation that is like the current one, except the patch level got the highest bit set, to keep continuity. The number was therefore 0x0090581f.

For backward compatibility, SSLEAY_VERSION_NUMBER is also defined.

SSLeay() returns this number. The return value can be compared to the macro to make sure that the correct version of the library has been loaded, especially when using DLLs on Windows systems.

SSLeay_version() returns different strings depending on **t**:

SSLEAY_VERSION
: The text variant of the version number and the release date. For example, ‘‘OpenSSL 0.9.5a 1 Apr 2000’’.

SSLEAY_CFLAGS
: The compiler flags set for the compilation process in the form ‘‘compiler: ...’’ if available or ‘‘compiler: information not available’’ otherwise.

SSLEAY_BUILT_ON
: The date of the build process in the form ‘‘built on: ...’’ if available or ‘‘built on: date not available’’ otherwise.

SSLEAY_PLATFORM
: The ‘‘Configure’’ target of the library build in the form ‘‘platform: ...’’ if available or ‘‘platform: information not available’’ otherwise.

SSLEAY_DIR
: The ‘‘OPENSSLDIR’’ setting of the library build in the form ‘‘OPENSSLDIR: ’’..."‘‘ if available or ’’OPENSSLDIR: N/A" otherwise.

For an unknown **t**, the text ‘‘not available’’ is returned.

RETURN VALUE

The version number.

SEE ALSO

crypto (3)

HISTORY

SSLeay() and SSLEAY_VERSION_NUMBER are available in all versions of SSLeay and OpenSSL. OPENSSL_VERSION_NUMBER is available in all versions of OpenSSL. **SSLEAY_DIR** was added in OpenSSL 0.9.7.

NAME

OPENSSL_config, OPENSSL_no_config − simple OpenSSL configuration functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/conf.h>

void OPENSSL_config(const char *config_name);
void OPENSSL_no_config(void);
```

DESCRIPTION

OPENSSL_config() configures OpenSSL using the standard **openssl.cnf** configuration file name using **config_name**. If **config_name** is NULL then the default name **openssl_conf** will be used. Any errors are ignored. Further calls to *OPENSSL_config()* will have no effect. The configuration file format is documented in the *conf*(5) manual page.

OPENSSL_no_config() disables configuration. If called before *OPENSSL_config()* no configuration takes place.

NOTES

It is **strongly** recommended that **all** new applications call *OPENSSL_config()* or the more sophisticated functions such as *CONF_modules_load()* during initialization (that is before starting any threads). By doing this an application does not need to keep track of all configuration options and some new functionality can be supported automatically.

It is also possible to automatically call *OPENSSL_config()* when an application calls *OPENSSL_add_all_algorithms()* by compiling an application with the preprocessor symbol **OPENSSL_LOAD_CONF** #define'd. In this way configuration can be added without source changes.

The environment variable **OPENSSL_CONF** can be set to specify the location of the configuration file.

Currently ASN1 OBJECTs and ENGINE configuration can be performed future versions of OpenSSL will add new configuration options.

There are several reasons why calling the OpenSSL configuration routines is advisable. For example new ENGINE functionality was added to OpenSSL 0.9.7. In OpenSSL 0.9.7 control functions can be supported by ENGINEs, this can be used (among other things) to load dynamic ENGINEs from shared libraries (DSOs). However very few applications currently support the control interface and so very few can load and use dynamic ENGINEs. Equally in future more sophisticated ENGINEs will require certain control operations to customize them. If an application calls *OPENSSL_config()* it doesn't need to know or care about ENGINE control operations because they can be performed by editing a configuration file.

Applications should free up configuration at application closedown by calling *CONF_modules_free()*.

RESTRICTIONS

The *OPENSSL_config()* function is designed to be a very simple ''call it and forget it'' function. As a result its behaviour is somewhat limited. It ignores all errors silently and it can only load from the standard configuration file location for example.

It is however **much** better than nothing. Applications which need finer control over their configuration functionality should use the configuration functions such as *CONF_load_modules()* directly.

RETURN VALUES

Neither *OPENSSL_config()* nor *OPENSSL_no_config()* return a value.

SEE ALSO

conf(5), *CONF_load_modules_file*(3), *CONF_modules_free*(3),*CONF_modules_free*(3)

HISTORY

OPENSSL_config() and *OPENSSL_no_config()* first appeared in OpenSSL 0.9.7

NAME

OPENSSL_ia32cap − finding the IA−32 processor capabilities

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
unsigned long *OPENSSL_ia32cap_loc(void);
#define OPENSSL_ia32cap (*(OPENSSL_ia32cap_loc()))
```

DESCRIPTION

Value returned by *OPENSSL_ia32cap_loc()* is address of a variable containing IA−32 processor capabilities bit vector as it appears in EDX register after executing CPUID instruction with EAX=1 input value (see Intel Application Note #241618). Naturally it's meaningful on IA−32[E] platforms only. The variable is normally set up automatically upon toolkit initialization, but can be manipulated afterwards to modify crypto library behaviour. For the moment of this writing six bits are significant, namely:

1. bit #28 denoting Hyperthreading, which is used to distiguish
cores with shared cache; 2. bit #26 denoting SSE2 support; 3. bit #25 denoting SSE support; 4. bit #23 denoting MMX support; 5. bit #20, reserved by Intel, is used to choose between RC4 code
pathes; 6. bit #4 denoting presence of Time-Stamp Counter.

For example, clearing bit #26 at run-time disables high-performance SSE2 code present in the crypto library. You might have to do this if target OpenSSL application is executed on SSE2 capable CPU, but under control of OS which does not support SSE2 extentions. Even though you can manipulate the value programmatically, you most likely will find it more appropriate to set up an environment variable with the same name prior starting target application, e.g. on Intel P4 processor 'env OPENSSL_ia32cap=0x12900010 apps/openssl', to achieve same effect without modifying the application source code. Alternatively you can reconfigure the toolkit with no−sse2 option and recompile.

NAME

OPENSSL_load_builtin_modules − add standard configuration modules

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/conf.h>

void OPENSSL_load_builtin_modules(void);
void ASN1_add_oid_module(void);
ENGINE_add_conf_module();
```

DESCRIPTION

The function *OPENSSL_load_builtin_modules()* adds all the standard OpenSSL configuration modules to the internal list. They can then be used by the OpenSSL configuration code.

ASN1_add_oid_module() adds just the ASN1 OBJECT module.

ENGINE_add_conf_module() adds just the ENGINE configuration module.

NOTES

If the simple configuration function *OPENSSL_config()* is called then *OPENSSL_load_builtin_modules()* is called automatically.

Applications which use the configuration functions directly will need to call *OPENSSL_load_builtin_modules()* themselves *before* any other configuration code.

Applications should call *OPENSSL_load_builtin_modules()* to load all configuration modules instead of adding modules selectively: otherwise functionality may be missing from the application if an when new modules are added.

RETURN VALUE

None of the functions return a value.

SEE ALSO

conf (3), *OPENSSL_config* (3)

HISTORY

These functions first appeared in OpenSSL 0.9.7.

NAME

OpenSSL_add_all_algorithms, OpenSSL_add_all_ciphers, OpenSSL_add_all_digests − add algorithms to internal table

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>

void OpenSSL_add_all_algorithms(void);
void OpenSSL_add_all_ciphers(void);
void OpenSSL_add_all_digests(void);

void EVP_cleanup(void);
```

DESCRIPTION

OpenSSL keeps an internal table of digest algorithms and ciphers. It uses this table to lookup ciphers via functions such as *EVP_get_cipher_byname()*.

OpenSSL_add_all_digests() adds all digest algorithms to the table.

OpenSSL_add_all_algorithms() adds all algorithms to the table (digests and ciphers).

OpenSSL_add_all_ciphers() adds all encryption algorithms to the table including password based encryption algorithms.

EVP_cleanup() removes all ciphers and digests from the table.

RETURN VALUES

None of the functions return a value.

NOTES

A typical application will call *OpenSSL_add_all_algorithms()* initially and *EVP_cleanup()* before exiting.

An application does not need to add algorithms to use them explicitly, for example by *EVP_sha1()*. It just needs to add them if it (or any of the functions it calls) needs to lookup algorithms.

The cipher and digest lookup functions are used in many parts of the library. If the table is not initialized several functions will misbehave and complain they cannot find algorithms. This includes the PEM, PKCS#12, SSL and S/MIME libraries. This is a common query in the OpenSSL mailing lists.

Calling *OpenSSL_add_all_algorithms()* links in all algorithms: as a result a statically linked executable can be quite large. If this is important it is possible to just add the required ciphers and digests.

BUGS

Although the functions do not return error codes it is possible for them to fail. This will only happen as a result of a memory allocation failure so this is not too much of a problem in practice.

SEE ALSO

openssl_evp (3), *EVP_DigestInit* (3), *EVP_EncryptInit* (3)

NAME

PKCS12_create − create a PKCS#12 structure

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/pkcs12.h>

PKCS12 *PKCS12_create(char *pass, char *name, EVP_PKEY *pkey, X509 *cert, STACK_OF(
                                  int nid_key, int nid_cert, int iter, int mac_iter, i
```

DESCRIPTION

PKCS12_create() creates a PKCS#12 structure.

pass is the passphrase to use. **name** is the **friendlyName** to use for the supplied certifictate and key. **pkey** is the private key to include in the structure and **cert** its corresponding certificates. **ca**, if not **NULL** is an optional set of certificates to also include in the structure.

nid_key and **nid_cert** are the encryption algorithms that should be used for the key and certificate respectively. **iter** is the encryption algorithm iteration count to use and **mac_iter** is the MAC iteration count to use. **keytype** is the type of key.

NOTES

The parameters **nid_key**, **nid_cert**, **iter**, **mac_iter** and **keytype** can all be set to zero and sensible defaults will be used.

These defaults are: 40 bit RC2 encryption for certificates, triple DES encryption for private keys, a key iteration count of PKCS12_DEFAULT_ITER (currently 2048) and a MAC iteration count of 1.

The default MAC iteration count is 1 in order to retain compatibility with old software which did not interpret MAC iteration counts. If such compatibility is not required then **mac_iter** should be set to PKCS12_DEFAULT_ITER.

keytype adds a flag to the store private key. This is a non standard extension that is only currently interpreted by MSIE. If set to zero the flag is omitted, if set to **KEY_SIG** the key can be used for signing only, if set to **KEY_EX** it can be used for signing and encryption. This option was useful for old export grade software which could use signing only keys of arbitrary size but had restrictions on the permissible sizes of keys which could be used for encryption.

NEW FUNCTIONALITY IN OPENSSL 0.9.8

Some additional functionality was added to *PKCS12_create()* in OpenSSL 0.9.8. These extensions are detailed below.

If a certificate contains an **alias** or **keyid** then this will be used for the corresponding **friendlyName** or **localKeyID** in the PKCS12 structure.

Either **pkey**, **cert** or both can be **NULL** to indicate that no key or certficate is required. In previous versions both had to be present or a fatal error is returned.

nid_key or **nid_cert** can be set to −1 indicating that no encryption should be used.

mac_iter can be set to −1 and the MAC will then be omitted entirely.

SEE ALSO

d2i_PKCS12 (3)

HISTORY

PKCS12_create was added in OpenSSL 0.9.3

NAME

PKCS12_parse − parse a PKCS#12 structure

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/pkcs12.h>
```

int PKCS12_parse(PKCS12 *p12, const char *pass, EVP_PKEY **pkey, X509 **cert, STACK_OF(X509) **ca);

DESCRIPTION

PKCS12_parse() parses a PKCS12 structure.

p12 is the **PKCS12** structure to parse. **pass** is the passphrase to use. If successful the private key will be written to ***pkey**, the corresponding certificate to ***cert** and any additional certificates to ***ca**.

NOTES

The parameters **pkey** and **cert** cannot be **NULL**. **ca** can be <NULL> in which case additional certificates will be discarded. ***ca** can also be a valid STACK in which case additional certificates are appended to ***ca**. If ***ca** is **NULL** a new STACK will be allocated.

The **friendlyName** and **localKeyID** attributes (if present) on each certificate will be stored in the **alias** and **keyid** attributes of the **X509** structure.

BUGS

Only a single private key and corresponding certificate is returned by this function. More complex PKCS#12 files with multiple private keys will only return the first match.

Only **friendlyName** and **localKeyID** attributes are currently stored in certificates. Other attributes are discarded.

Attributes currently cannot be store in the private key **EVP_PKEY** structure.

SEE ALSO

d2i_PKCS12 (3)

HISTORY

PKCS12_parse was added in OpenSSL 0.9.3

NAME

PKCS7_decrypt − decrypt content from a PKCS#7 envelopedData structure

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/pkcs7.h>

int PKCS7_decrypt(PKCS7 *p7, EVP_PKEY *pkey, X509 *cert, BIO *data, int flags);
```

DESCRIPTION

PKCS7_decrypt() extracts and decrypts the content from a PKCS#7 envelopedData structure. **pkey** is the private key of the recipient, **cert** is the recipients certificate, **data** is a BIO to write the content to and **flags** is an optional set of flags.

NOTES

OpenSSL_add_all_algorithms() (or equivalent) should be called before using this function or errors about unknown algorithms will occur.

Although the recipients certificate is not needed to decrypt the data it is needed to locate the appropriate (of possible several) recipients in the PKCS#7 structure.

The following flags can be passed in the **flags** parameter.

If the **PKCS7_TEXT** flag is set MIME headers for type **text/plain** are deleted from the content. If the content is not of type **text/plain** then an error is returned.

RETURN VALUES

PKCS7_decrypt() returns either 1 for success or 0 for failure. The error can be obtained from *ERR_get_error* (3)

BUGS

PKCS7_decrypt() must be passed the correct recipient key and certificate. It would be better if it could look up the correct key and certificate from a database.

The lack of single pass processing and need to hold all data in memory as mentioned in *PKCS7_sign()* also applies to *PKCS7_verify()*.

SEE ALSO

ERR_get_error (3), *PKCS7_encrypt* (3)

HISTORY

PKCS7_decrypt() was added to OpenSSL 0.9.5

NAME

PKCS7_encrypt − create a PKCS#7 envelopedData structure

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/pkcs7.h>

PKCS7 *PKCS7_encrypt(STACK_OF(X509) *certs, BIO *in, const EVP_CIPHER *cipher, int
```

DESCRIPTION

PKCS7_encrypt() creates and returns a PKCS#7 envelopedData structure. **certs** is a list of recipient certificates. **in** is the content to be encrypted. **cipher** is the symmetric cipher to use. **flags** is an optional set of flags.

NOTES

Only RSA keys are supported in PKCS#7 and envelopedData so the recipient certificates supplied to this function must all contain RSA public keys, though they do not have to be signed using the RSA algorithm.

EVP_des_ede3_cbc() (triple DES) is the algorithm of choice for S/MIME use because most clients will support it.

Some old ''export grade'' clients may only support weak encryption using 40 or 64 bit RC2. These can be used by passing *EVP_rc2_40_cbc()* and *EVP_rc2_64_cbc()* respectively.

The algorithm passed in the **cipher** parameter must support ASN1 encoding of its parameters.

Many browsers implement a ''sign and encrypt'' option which is simply an S/MIME envelopedData containing an S/MIME signed message. This can be readily produced by storing the S/MIME signed message in a memory BIO and passing it to *PKCS7_encrypt()*.

The following flags can be passed in the **flags** parameter.

If the **PKCS7_TEXT** flag is set MIME headers for type **text/plain** are prepended to the data.

Normally the supplied content is translated into MIME canonical format (as required by the S/MIME specifications) if **PKCS7_BINARY** is set no translation occurs. This option should be used if the supplied data is in binary format otherwise the translation will corrupt it. If **PKCS7_BINARY** is set then **PKCS7_TEXT** is ignored.

If the **PKCS7_STREAM** flag is set a partial **PKCS7** structure is output suitable for streaming I/O: no data is read from the BIO **in**.

NOTES

If the flag **PKCS7_STREAM** is set the returned **PKCS7** structure is **not** complete and outputting its contents via a function that does not properly finalize the **PKCS7** structure will give unpredictable results.

Several functions including *SMIME_write_PKCS7()*, *i2d_PKCS7_bio_stream()*, *PEM_write_bio_PKCS7_stream()* finalize the structure. Alternatively finalization can be performed by obtaining the streaming ASN1 **BIO** directly using *BIO_new_PKCS7()*.

RETURN VALUES

PKCS7_encrypt() returns either a PKCS7 structure or NULL if an error occurred. The error can be obtained from *ERR_get_error* (3).

SEE ALSO

ERR_get_error (3), *PKCS7_decrypt* (3)

HISTORY

PKCS7_decrypt() was added to OpenSSL 0.9.5 The **PKCS7_STREAM** flag was first supported in OpenSSL 0.9.9.

NAME

PKCS7_sign − create a PKCS#7 signedData structure

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/pkcs7.h>

PKCS7 *PKCS7_sign(X509 *signcert, EVP_PKEY *pkey, STACK_OF(X509) *certs, BIO *data,
```

DESCRIPTION

PKCS7_sign() creates and returns a PKCS#7 signedData structure. **signcert** is the certificate to sign with, **pkey** is the corresponsding private key. **certs** is an optional additional set of certificates to include in the PKCS#7 structure (for example any intermediate CAs in the chain).

The data to be signed is read from BIO **data**.

flags is an optional set of flags.

NOTES

Any of the following flags (ored together) can be passed in the **flags** parameter.

Many S/MIME clients expect the signed content to include valid MIME headers. If the **PKCS7_TEXT** flag is set MIME headers for type **text/plain** are prepended to the data.

If **PKCS7_NOCERTS** is set the signer's certificate will not be included in the PKCS7 structure, the signer's certificate must still be supplied in the **signcert** parameter though. This can reduce the size of the signature if the signers certificate can be obtained by other means: for example a previously signed message.

The data being signed is included in the PKCS7 structure, unless **PKCS7_DETACHED** is set in which case it is omitted. This is used for PKCS7 detached signatures which are used in S/MIME plaintext signed messages for example.

Normally the supplied content is translated into MIME canonical format (as required by the S/MIME specifications) if **PKCS7_BINARY** is set no translation occurs. This option should be used if the supplied data is in binary format otherwise the translation will corrupt it.

The signedData structure includes several PKCS#7 autenticatedAttributes including the signing time, the PKCS#7 content type and the supported list of ciphers in an SMIMECapabilities attribute. If **PKCS7_NOATTR** is set then no authenticatedAttributes will be used. If **PKCS7_NOSMIMECAP** is set then just the SMIMECapabilities are omitted.

If present the SMIMECapabilities attribute indicates support for the following algorithms: triple DES, 128 bit RC2, 64 bit RC2, DES and 40 bit RC2. If any of these algorithms is disabled then it will not be included.

If the flags **PKCS7_STREAM** is set then the returned **PKCS7** structure is just initialized ready to perform the signing operation. The signing is however **not** performed and the data to be signed is not read from the **data** parameter. Signing is deferred until after the data has been written. In this way data can be signed in a single pass.

If the **PKCS7_PARTIAL** flag is set a partial **PKCS7** structure is output to which additional signers and capabilities can be added before finalization.

NOTES

If the flag **PKCS7_STREAM** is set the returned **PKCS7** structure is **not** complete and outputting its contents via a function that does not properly finalize the **PKCS7** structure will give unpredictable results.

Several functions including *SMIME_write_PKCS7(), i2d_PKCS7_bio_stream(), PEM_write_bio_PKCS7_stream()* finalize the structure. Alternatively finalization can be performed by obtaining the streaming ASN1 **BIO** directly using *BIO_new_PKCS7()*.

If a signer is specified it will use the default digest for the signing algorithm. This is **SHA1** for both RSA and DSA keys.

In OpenSSL 0.9.9 the **certs**, **signcert** and **pkey** parameters can all be **NULL** if the **PKCS7_PARTIAL** flag is

set. One or more signers can be added using the function ***PKCS7_sign_add_signer()***. ***PKCS7_final()*** must also be called to finalize the structure if streaming is not enabled. Alternative signing digests can also be specified using this method.

In OpenSSL 0.9.9 if **signcert** and **pkey** are NULL then a certificates only PKCS#7 structure is output.

In versions of OpenSSL before 0.9.9 the **signcert** and **pkey** parameters must **NOT** be NULL.

BUGS

Some advanced attributes such as counter signatures are not supported.

RETURN VALUES

PKCS7_sign() returns either a valid PKCS7 structure or NULL if an error occurred. The error can be obtained from *ERR_get_error* (3).

SEE ALSO

ERR_get_error (3), *PKCS7_verify* (3)

HISTORY

PKCS7_sign() was added to OpenSSL 0.9.5

The **PKCS7_PARTIAL** flag was added in OpenSSL 0.9.9

The **PKCS7_STREAM** flag was added in OpenSSL 0.9.9

NAME

PKCS7_verify − verify a PKCS#7 signedData structure

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/pkcs7.h>

int PKCS7_verify(PKCS7 *p7, STACK_OF(X509) *certs, X509_STORE *store, BIO *indata,

STACK_OF(X509) *PKCS7_get0_signers(PKCS7 *p7, STACK_OF(X509) *certs, int flags);
```

DESCRIPTION

PKCS7_verify() verifies a PKCS#7 signedData structure. **p7** is the PKCS7 structure to verify. **certs** is a set of certificates in which to search for the signer's certificate. **store** is a trusted certficate store (used for chain verification). **indata** is the signed data if the content is not present in **p7** (that is it is detached). The content is written to **out** if it is not NULL.

flags is an optional set of flags, which can be used to modify the verify operation.

PKCS7_get0_signers() retrieves the signer's certificates from **p7**, it does **not** check their validity or whether any signatures are valid. The **certs** and **flags** parameters have the same meanings as in *PKCS7_verify()*.

VERIFY PROCESS

Normally the verify process proceeds as follows.

Initially some sanity checks are performed on **p7**. The type of **p7** must be signedData. There must be at least one signature on the data and if the content is detached **indata** cannot be **NULL**.

An attempt is made to locate all the signer's certificates, first looking in the **certs** parameter (if it is not **NULL**) and then looking in any certificates contained in the **p7** structure itself. If any signer's certificates cannot be located the operation fails.

Each signer's certificate is chain verified using the **smimesign** purpose and the supplied trusted certificate store. Any internal certificates in the message are used as untrusted CAs. If any chain verify fails an error code is returned.

Finally the signed content is read (and written to **out** is it is not NULL) and the signature's checked.

If all signature's verify correctly then the function is successful.

Any of the following flags (ored together) can be passed in the **flags** parameter to change the default verify behaviour. Only the flag **PKCS7_NOINTERN** is meaningful to *PKCS7_get0_signers()*.

If **PKCS7_NOINTERN** is set the certificates in the message itself are not searched when locating the signer's certificate. This means that all the signers certificates must be in the **certs** parameter.

If the **PKCS7_TEXT** flag is set MIME headers for type **text/plain** are deleted from the content. If the content is not of type **text/plain** then an error is returned.

If **PKCS7_NOVERIFY** is set the signer's certificates are not chain verified.

If **PKCS7_NOCHAIN** is set then the certificates contained in the message are not used as untrusted CAs. This means that the whole verify chain (apart from the signer's certificate) must be contained in the trusted store.

If **PKCS7_NOSIGS** is set then the signatures on the data are not checked.

NOTES

One application of **PKCS7_NOINTERN** is to only accept messages signed by a small number of certificates. The acceptable certificates would be passed in the **certs** parameter. In this case if the signer is not one of the certificates supplied in **certs** then the verify will fail because the signer cannot be found.

Care should be taken when modifying the default verify behaviour, for example setting **PKCS7_NOVERIFY|PKCS7_NOSIGS** will totally disable all verification and any signed message will be considered valid. This combination is however useful if one merely wishes to write the content to **out** and its validity is

not considered important.

Chain verification should arguably be performed using the signing time rather than the current time. However since the signing time is supplied by the signer it cannot be trusted without additional evidence (such as a trusted timestamp).

RETURN VALUES

PKCS7_verify() returns 1 for a successful verification and zero or a negative value if an error occurs.

PKCS7_get0_signers() returns all signers or **NULL** if an error occurred.

The error can be obtained from *ERR_get_error* (3)

BUGS

The trusted certificate store is not searched for the signers certificate, this is primarily due to the inadequacies of the current **X509_STORE** functionality.

The lack of single pass processing and need to hold all data in memory as mentioned in *PKCS7_sign()* also applies to *PKCS7_verify()*.

SEE ALSO

ERR_get_error (3), *PKCS7_sign* (3)

HISTORY

PKCS7_verify() was added to OpenSSL 0.9.5

NAME

RAND_add, RAND_seed, RAND_status, RAND_event, RAND_screen − add entropy to the PRNG

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rand.h>

void RAND_seed(const void *buf, int num);

void RAND_add(const void *buf, int num, double entropy);

int  RAND_status(void);

int  RAND_event(UINT iMsg, WPARAM wParam, LPARAM lParam);
void RAND_screen(void);
```

DESCRIPTION

RAND_add() mixes the **num** bytes at **buf** into the PRNG state. Thus, if the data at **buf** are unpredictable to an adversary, this increases the uncertainty about the state and makes the PRNG output less predictable. Suitable input comes from user interaction (random key presses, mouse movements) and certain hardware events. The **entropy** argument is (the lower bound of) an estimate of how much randomness is contained in **buf**, measured in bytes. Details about sources of randomness and how to estimate their entropy can be found in the literature, e.g. RFC 1750.

RAND_add() may be called with sensitive data such as user entered passwords. The seed values cannot be recovered from the PRNG output.

OpenSSL makes sure that the PRNG state is unique for each thread. On systems that provide `/dev/urandom`, the randomness device is used to seed the PRNG transparently. However, on all other systems, the application is responsible for seeding the PRNG by calling *RAND_add()*, *RAND_egd* (3) or *RAND_load_file* (3).

RAND_seed() is equivalent to *RAND_add()* when **num** == **entropy**.

RAND_event() collects the entropy from Windows events such as mouse movements and other user interaction. It should be called with the **iMsg**, **wParam** and **lParam** arguments of *all* messages sent to the window procedure. It will estimate the entropy contained in the event message (if any), and add it to the PRNG. The program can then process the messages as usual.

The *RAND_screen()* function is available for the convenience of Windows programmers. It adds the current contents of the screen to the PRNG. For applications that can catch Windows events, seeding the PRNG by calling *RAND_event()* is a significantly better source of randomness. It should be noted that both methods cannot be used on servers that run without user interaction.

RETURN VALUES

RAND_status() and *RAND_event()* return 1 if the PRNG has been seeded with enough data, 0 otherwise.

The other functions do not return values.

SEE ALSO

openssl_rand (3), *RAND_egd* (3), *RAND_load_file* (3), *RAND_cleanup* (3)

HISTORY

RAND_seed() and *RAND_screen()* are available in all versions of SSLeay and OpenSSL. *RAND_add()* and *RAND_status()* have been added in OpenSSL 0.9.5, *RAND_event()* in OpenSSL 0.9.5a.

NAME

RAND_bytes, RAND_pseudo_bytes − generate random data

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rand.h>

int RAND_bytes(unsigned char *buf, int num);

int RAND_pseudo_bytes(unsigned char *buf, int num);
```

DESCRIPTION

RAND_bytes() puts **num** cryptographically strong pseudo-random bytes into **buf**. An error occurs if the PRNG has not been seeded with enough randomness to ensure an unpredictable byte sequence.

RAND_pseudo_bytes() puts **num** pseudo-random bytes into **buf**. Pseudo-random byte sequences generated by *RAND_pseudo_bytes()* will be unique if they are of sufficient length, but are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

The contents of **buf** is mixed into the entropy pool before retrieving the new pseudo-random bytes unless disabled at compile time (see FAQ).

RETURN VALUES

RAND_bytes() returns 1 on success, 0 otherwise. The error code can be obtained by *ERR_get_error* (3). *RAND_pseudo_bytes()* returns 1 if the bytes generated are cryptographically strong, 0 otherwise. Both functions return −1 if they are not supported by the current RAND method.

SEE ALSO

openssl_rand (3), *ERR_get_error* (3), *RAND_add* (3)

HISTORY

RAND_bytes() is available in all versions of SSLeay and OpenSSL. It has a return value since OpenSSL 0.9.5. *RAND_pseudo_bytes()* was added in OpenSSL 0.9.5.

NAME

RAND_cleanup − erase the PRNG state

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rand.h>

void RAND_cleanup(void);
```

DESCRIPTION

RAND_cleanup() erases the memory used by the PRNG.

RETURN VALUE

RAND_cleanup() returns no value.

SEE ALSO

openssl_rand (3)

HISTORY

RAND_cleanup() is available in all versions of SSLeay and OpenSSL.

NAME

RAND_egd − query entropy gathering daemon

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rand.h>

int RAND_egd(const char *path);
int RAND_egd_bytes(const char *path, int bytes);

int RAND_query_egd_bytes(const char *path, unsigned char *buf, int bytes);
```

DESCRIPTION

RAND_egd() queries the entropy gathering daemon EGD on socket **path**. It queries 255 bytes and uses *RAND_add* (3) to seed the OpenSSL built-in PRNG. RAND_egd(path) is a wrapper for RAND_egd_bytes(path, 255);

RAND_egd_bytes() queries the entropy gathering daemon EGD on socket **path**. It queries **bytes** bytes and uses *RAND_add* (3) to seed the OpenSSL built-in PRNG. This function is more flexible than *RAND_egd()*. When only one secret key must be generated, it is not necessary to request the full amount 255 bytes from the EGD socket. This can be advantageous, since the amount of entropy that can be retrieved from EGD over time is limited.

RAND_query_egd_bytes() performs the actual query of the EGD daemon on socket **path**. If **buf** is given, **bytes** bytes are queried and written into **buf**. If **buf** is NULL, **bytes** bytes are queried and used to seed the OpenSSL built-in PRNG using *RAND_add* (3).

NOTES

On systems without /dev/*random devices providing entropy from the kernel, the EGD entropy gathering daemon can be used to collect entropy. It provides a socket interface through which entropy can be gathered in chunks up to 255 bytes. Several chunks can be queried during one connection.

EGD is available from http://www.lothar.com/tech/crypto/ (`perl Makefile.PL; make; make install` to install). It is run as **egd** *path*, where *path* is an absolute path designating a socket. When *RAND_egd()* is called with that path as an argument, it tries to read random bytes that EGD has collected. The read is performed in non-blocking mode.

Alternatively, the EGD-interface compatible daemon PRNGD can be used. It is available from http://prngd.sourceforge.net/ . PRNGD does employ an internal PRNG itself and can therefore never run out of entropy.

OpenSSL automatically queries EGD when entropy is requested via *RAND_bytes()* or the status is checked via *RAND_status()* for the first time, if the socket is located at /var/run/egd−pool, /dev/egd−pool or /etc/egd−pool.

RETURN VALUE

RAND_egd() and *RAND_egd_bytes()* return the number of bytes read from the daemon on success, and −1 if the connection failed or the daemon did not return enough data to fully seed the PRNG.

RAND_query_egd_bytes() returns the number of bytes read from the daemon on success, and −1 if the connection failed. The PRNG state is not considered.

SEE ALSO

openssl_rand (3), *RAND_add* (3), *RAND_cleanup* (3)

HISTORY

RAND_egd() is available since OpenSSL 0.9.5.

RAND_egd_bytes() is available since OpenSSL 0.9.6.

RAND_query_egd_bytes() is available since OpenSSL 0.9.7.

The automatic query of /var/run/egd−pool et al was added in OpenSSL 0.9.7.

NAME

RAND_load_file, RAND_write_file, RAND_file_name − PRNG seed file

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rand.h>

const char *RAND_file_name(char *buf, size_t num);

int RAND_load_file(const char *filename, long max_bytes);

int RAND_write_file(const char *filename);
```

DESCRIPTION

RAND_file_name() generates a default path for the random seed file. **buf** points to a buffer of size **num** in which to store the filename. The seed file is `$RANDFILE` if that environment variable is set, `$HOME`/.rnd otherwise. If `$HOME` is not set either, or **num** is too small for the path name, an error occurs.

RAND_load_file() reads a number of bytes from file **filename** and adds them to the PRNG. If **max_bytes** is non−negative, up to to **max_bytes** are read; starting with OpenSSL 0.9.5, if **max_bytes** is −1, the complete file is read.

RAND_write_file() writes a number of random bytes (currently 1024) to file **filename** which can be used to initialize the PRNG by calling *RAND_load_file()* in a later session.

RETURN VALUES

RAND_load_file() returns the number of bytes read.

RAND_write_file() returns the number of bytes written, and −1 if the bytes written were generated without appropriate seed.

RAND_file_name() returns a pointer to **buf** on success, and NULL on error.

SEE ALSO

openssl_rand (3), *RAND_add* (3), *RAND_cleanup* (3)

HISTORY

RAND_load_file(), *RAND_write_file()* and *RAND_file_name()* are available in all versions of SSLeay and OpenSSL.

NAME

RAND_set_rand_method, RAND_get_rand_method, RAND_SSLeay − select RAND method

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rand.h>

void RAND_set_rand_method(const RAND_METHOD *meth);

const RAND_METHOD *RAND_get_rand_method(void);

RAND_METHOD *RAND_SSLeay(void);
```

DESCRIPTION

A **RAND_METHOD** specifies the functions that OpenSSL uses for random number generation. By modifying the method, alternative implementations such as hardware RNGs may be used. IMPORTANT: See the NOTES section for important information about how these RAND API functions are affected by the use of **ENGINE** API calls.

Initially, the default RAND_METHOD is the OpenSSL internal implementation, as returned by *RAND_SSLeay()*.

RAND_set_default_method() makes **meth** the method for PRNG use. **NB**: This is true only whilst no ENGINE has been set as a default for RAND, so this function is no longer recommended.

RAND_get_default_method() returns a pointer to the current RAND_METHOD. However, the meaningfulness of this result is dependent on whether the ENGINE API is being used, so this function is no longer recommended.

THE RAND_METHOD STRUCTURE

```
typedef struct rand_meth_st
{
       void (*seed)(const void *buf, int num);
       int (*bytes)(unsigned char *buf, int num);
       void (*cleanup)(void);
       void (*add)(const void *buf, int num, int entropy);
       int (*pseudorand)(unsigned char *buf, int num);
       int (*status)(void);
} RAND_METHOD;
```

The components point to the implementation of *RAND_seed()*, *RAND_bytes()*, *RAND_cleanup()*, *RAND_add()*, *RAND_pseudo_rand()* and *RAND_status()*. Each component may be NULL if the function is not implemented.

RETURN VALUES

RAND_set_rand_method() returns no value. *RAND_get_rand_method()* and *RAND_SSLeay()* return pointers to the respective methods.

NOTES

As of version 0.9.7, RAND_METHOD implementations are grouped together with other algorithmic APIs (eg. RSA_METHOD, EVP_CIPHER, etc) in **ENGINE** modules. If a default ENGINE is specified for RAND functionality using an ENGINE API function, that will override any RAND defaults set using the RAND API (ie. *RAND_set_rand_method()*). For this reason, the ENGINE API is the recommended way to control default implementations for use in RAND and other cryptographic algorithms.

SEE ALSO

openssl_rand (3), *engine* (3)

HISTORY

RAND_set_rand_method(), *RAND_get_rand_method()* and *RAND_SSLeay()* are available in all versions of OpenSSL.

In the engine version of version 0.9.6, *RAND_set_rand_method()* was altered to take an ENGINE pointer as its argument. As of version 0.9.7, that has been reverted as the ENGINE API transparently overrides RAND defaults if used, otherwise RAND API functions work as before. *RAND_set_rand_engine()* was also introduced in version 0.9.7.

NAME

RSA_blinding_on, RSA_blinding_off − protect the RSA operation from timing attacks

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_blinding_on(RSA *rsa, BN_CTX *ctx);

void RSA_blinding_off(RSA *rsa);
```

DESCRIPTION

RSA is vulnerable to timing attacks. In a setup where attackers can measure the time of RSA decryption or signature operations, blinding must be used to protect the RSA operation from that attack.

RSA_blinding_on() turns blinding on for key **rsa** and generates a random blinding factor. **ctx** is **NULL** or a pre-allocated and initialized **BN_CTX**. The random number generator must be seeded prior to calling *RSA_blinding_on()*.

RSA_blinding_off() turns blinding off and frees the memory used for the blinding factor.

RETURN VALUES

RSA_blinding_on() returns 1 on success, and 0 if an error occurred.

RSA_blinding_off() returns no value.

SEE ALSO

openssl_rsa (3), *openssl_rand* (3)

HISTORY

RSA_blinding_on() and *RSA_blinding_off()* appeared in SSLeay 0.9.0.

NAME

RSA_check_key − validate private RSA keys

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_check_key(RSA *rsa);
```

DESCRIPTION

This function validates RSA keys. It checks that **p** and **q** are in fact prime, and that **n = p*q**.

It also checks that **d*e = 1 mod (p−1*q−1)**, and that **dmp1**, **dmq1** and **iqmp** are set correctly or are **NULL**.

As such, this function can not be used with any arbitrary RSA key object, even if it is otherwise fit for regular RSA operation. See **NOTES** for more information.

RETURN VALUE

RSA_check_key() returns 1 if **rsa** is a valid RSA key, and 0 otherwise. −1 is returned if an error occurs while checking the key.

If the key is invalid or an error occurred, the reason code can be obtained using *ERR_get_error* (3).

NOTES

This function does not work on RSA public keys that have only the modulus and public exponent elements populated. It performs integrity checks on all the RSA key material, so the RSA key structure must contain all the private key data too.

Unlike most other RSA functions, this function does **not** work transparently with any underlying ENGINE implementation because it uses the key data in the RSA structure directly. An ENGINE implementation can override the way key data is stored and handled, and can even provide support for HSM keys − in which case the RSA structure may contain **no** key data at all! If the ENGINE in question is only being used for acceleration or analysis purposes, then in all likelihood the RSA key data is complete and untouched, but this can't be assumed in the general case.

BUGS

A method of verifying the RSA key using opaque RSA API functions might need to be considered. Right now *RSA_check_key()* simply uses the RSA structure elements directly, bypassing the RSA_METHOD table altogether (and completely violating encapsulation and object-orientation in the process). The best fix will probably be to introduce a "*check_key()*" handler to the RSA_METHOD function table so that alternative implementations can also provide their own verifiers.

SEE ALSO

openssl_rsa (3), *ERR_get_error* (3)

HISTORY

RSA_check_key() appeared in OpenSSL 0.9.4.

NAME

RSA_generate_key − generate RSA key pair

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

RSA *RSA_generate_key(int num, unsigned long e,
   void (*callback)(int,int,void *), void *cb_arg);
```

DESCRIPTION

RSA_generate_key() generates a key pair and returns it in a newly allocated **RSA** structure. The pseudo-random number generator must be seeded prior to calling *RSA_generate_key()*.

The modulus size will be **num** bits, and the public exponent will be **e**. Key sizes with **num** < 1024 should be considered insecure. The exponent is an odd number, typically 3, 17 or 65537.

A callback function may be used to provide feedback about the progress of the key generation. If **callback** is not **NULL**, it will be called as follows:

- While a random prime number is generated, it is called as described in *BN_generate_prime* (3).
- When the n−th randomly generated prime is rejected as not suitable for the key, **callback(2, n, cb_arg)** is called.
- When a random p has been found with p−1 relatively prime to **e**, it is called as **callback(3, 0, cb_arg)**.

The process is then repeated for prime q with **callback(3, 1, cb_arg)**.

RETURN VALUE

If key generation fails, *RSA_generate_key()* returns **NULL**; the error codes can be obtained by *ERR_get_error* (3).

BUGS

callback(2, x, cb_arg) is used with two different meanings.

RSA_generate_key() goes into an infinite loop for illegal input values.

SEE ALSO

ERR_get_error (3), *openssl_rand* (3), *openssl_rsa* (3), *RSA_free* (3)

HISTORY

The **cb_arg** argument was added in SSLeay 0.9.0.

NAME

RSA_get_ex_new_index, RSA_set_ex_data, RSA_get_ex_data − add application specific data to RSA structures

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_get_ex_new_index(long argl, void *argp,
                         CRYPTO_EX_new *new_func,
                         CRYPTO_EX_dup *dup_func,
                         CRYPTO_EX_free *free_func);

int RSA_set_ex_data(RSA *r, int idx, void *arg);

void *RSA_get_ex_data(RSA *r, int idx);

typedef int CRYPTO_EX_new(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
                          int idx, long argl, void *argp);
typedef void CRYPTO_EX_free(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
                            int idx, long argl, void *argp);
typedef int CRYPTO_EX_dup(CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d,
                          int idx, long argl, void *argp);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. This has several potential uses, it can be used to cache data associated with a structure (for example the hash of some part of the structure) or some additional data (for example a handle to the data in an external library).

Since the application data can be anything at all it is passed and retrieved as a **void *** type.

The ***RSA_get_ex_new_index()*** function is initially called to ''register'' some new application specific data. It takes three optional function pointers which are called when the parent structure (in this case an RSA structure) is initially created, when it is copied and when it is freed up. If any or all of these function pointer arguments are not used they should be set to NULL. The precise manner in which these function pointers are called is described in more detail below. ***RSA_get_ex_new_index()*** also takes additional long and pointer parameters which will be passed to the supplied functions but which otherwise have no special meaning. It returns an **index** which should be stored (typically in a static variable) and passed used in the **idx** parameter in the remaining functions. Each successful call to ***RSA_get_ex_new_index()*** will return an index greater than any previously returned, this is important because the optional functions are called in order of increasing index value.

RSA_set_ex_data() is used to set application specific data, the data is supplied in the **arg** parameter and its precise meaning is up to the application.

RSA_get_ex_data() is used to retrieve application specific data. The data is returned to the application, this will be the same value as supplied to a previous ***RSA_set_ex_data()*** call.

new_func() is called when a structure is initially allocated (for example with ***RSA_new()***. The parent structure members will not have any meaningful values at this point. This function will typically be used to allocate any application specific structure.

free_func() is called when a structure is being freed up. The dynamic parent structure members should not be accessed because they will be freed up when this function is called.

new_func() and ***free_func()*** take the same parameters. **parent** is a pointer to the parent RSA structure. **ptr** is a the application specific data (this wont be of much use in ***new_func()***. **ad** is a pointer to the **CRYPTO_EX_DATA** structure from the parent RSA structure: the functions ***CRYPTO_get_ex_data()*** and ***CRYPTO_set_ex_data()*** can be called to manipulate it. The **idx** parameter is the index: this will be the same value returned by ***RSA_get_ex_new_index()*** when the functions were initially registered. Finally the **argl** and **argp** parameters are the values originally passed to the same corresponding parameters when

RSA_get_ex_new_index() was called.

dup_func() is called when a structure is being copied. Pointers to the destination and source **CRYPTO_EX_DATA** structures are passed in the **to** and **from** parameters respectively. The **from_d** parameter is passed a pointer to the source application data when the function is called, when the function returns the value is copied to the destination: the application can thus modify the data pointed to by **from_d** and have different values in the source and destination. The **idx**, **argl** and **argp** parameters are the same as those in ***new_func()*** and ***free_func()***.

RETURN VALUES

RSA_get_ex_new_index() returns a new index or −1 on failure (note 0 is a valid index value).

RSA_set_ex_data() returns 1 on success or 0 on failure.

RSA_get_ex_data() returns the application data or 0 on failure. 0 may also be valid application data but currently it can only fail if given an invalid **idx** parameter.

new_func() and ***dup_func()*** should return 0 for failure and 1 for success.

On failure an error code can be obtained from *ERR_get_error* (3).

BUGS

dup_func() is currently never called.

The return value of ***new_func()*** is ignored.

The ***new_func()*** function isn't very useful because no meaningful values are present in the parent RSA structure when it is called.

SEE ALSO

openssl_rsa (3), *CRYPTO_set_ex_data* (3)

HISTORY

RSA_get_ex_new_index(), *RSA_set_ex_data()* and *RSA_get_ex_data()* are available since SSLeay 0.9.0.

NAME

RSA_new, RSA_free − allocate and free RSA objects

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

RSA * RSA_new(void);

void RSA_free(RSA *rsa);
```

DESCRIPTION

RSA_new() allocates and initializes an **RSA** structure. It is equivalent to calling RSA_new_method(NULL).

RSA_free() frees the **RSA** structure and its components. The key is erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, *RSA_new()* returns **NULL** and sets an error code that can be obtained by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

RSA_free() returns no value.

SEE ALSO

ERR_get_error (3), *openssl_rsa* (3), *RSA_generate_key* (3), *RSA_new_method* (3)

HISTORY

RSA_new() and *RSA_free()* are available in all versions of SSLeay and OpenSSL.

NAME

RSA_padding_add_PKCS1_type_1, RSA_padding_check_PKCS1_type_1, RSA_padding_add_PKCS1_type_2, RSA_padding_check_PKCS1_type_2, RSA_padding_add_PKCS1_OAEP, RSA_padding_check_PKCS1_OAEP, RSA_padding_add_SSLv23, RSA_padding_check_SSLv23, RSA_padding_add_none, RSA_padding_check_none − asymmetric encryption padding

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_padding_add_PKCS1_type_1(unsigned char *to, int tlen,
   unsigned char *f, int fl);

int RSA_padding_check_PKCS1_type_1(unsigned char *to, int tlen,
   unsigned char *f, int fl, int rsa_len);

int RSA_padding_add_PKCS1_type_2(unsigned char *to, int tlen,
   unsigned char *f, int fl);

int RSA_padding_check_PKCS1_type_2(unsigned char *to, int tlen,
   unsigned char *f, int fl, int rsa_len);

int RSA_padding_add_PKCS1_OAEP(unsigned char *to, int tlen,
   unsigned char *f, int fl, unsigned char *p, int pl);

int RSA_padding_check_PKCS1_OAEP(unsigned char *to, int tlen,
   unsigned char *f, int fl, int rsa_len, unsigned char *p, int pl);

int RSA_padding_add_SSLv23(unsigned char *to, int tlen,
   unsigned char *f, int fl);

int RSA_padding_check_SSLv23(unsigned char *to, int tlen,
   unsigned char *f, int fl, int rsa_len);

int RSA_padding_add_none(unsigned char *to, int tlen,
   unsigned char *f, int fl);

int RSA_padding_check_none(unsigned char *to, int tlen,
   unsigned char *f, int fl, int rsa_len);
```

DESCRIPTION

The *RSA_padding_xxx_xxx()* functions are called from the RSA encrypt, decrypt, sign and verify functions. Normally they should not be called from application programs.

However, they can also be called directly to implement padding for other asymmetric ciphers. *RSA_padding_add_PKCS1_OAEP()* and *RSA_padding_check_PKCS1_OAEP()* may be used in an application combined with **RSA_NO_PADDING** in order to implement OAEP with an encoding parameter.

RSA_padding_add_xxx() encodes **fl** bytes from **f** so as to fit into **tlen** bytes and stores the result at **to**. An error occurs if **fl** does not meet the size requirements of the encoding method.

The following encoding methods are implemented:

PKCS1_type_1
: PKCS #1 v2.0 EMSA−PKCS1−v1_5 (PKCS #1 v1.5 block type 1); used for signatures

PKCS1_type_2
: PKCS #1 v2.0 EME−PKCS1−v1_5 (PKCS #1 v1.5 block type 2)

PKCS1_OAEP
: PKCS #1 v2.0 EME-OAEP

SSLv23
: PKCS #1 EME−PKCS1−v1_5 with SSL-specific modification

none
: simply copy the data

The random number generator must be seeded prior to calling *RSA_padding_add_xxx()*.

RSA_padding_check_xxx() verifies that the **fl** bytes at **f** contain a valid encoding for a **rsa_len** byte RSA key in the respective encoding method and stores the recovered data of at most **tlen** bytes (for **RSA_NO_PADDING**: of size **tlen**) at **to**.

For *RSA_padding_xxx_OAEP()*, **p** points to the encoding parameter of length **pl**. **p** may be **NULL** if **pl** is 0.

RETURN VALUES

The *RSA_padding_add_xxx()* functions return 1 on success, 0 on error. The *RSA_padding_check_xxx()* functions return the length of the recovered data, −1 on error. Error codes can be obtained by calling *ERR_get_error* (3).

SEE ALSO

RSA_public_encrypt (3), *RSA_private_decrypt* (3), *RSA_sign* (3), *RSA_verify* (3)

HISTORY

RSA_padding_add_PKCS1_type_1(), *RSA_padding_check_PKCS1_type_1()*, *RSA_padding_add_PKCS1_type_2()*, *RSA_padding_check_PKCS1_type_2()*, *RSA_padding_add_SSLv23()*, *RSA_padding_check_SSLv23()*, *RSA_padding_add_none()* and *RSA_padding_check_none()* appeared in SSLeay 0.9.0.

RSA_padding_add_PKCS1_OAEP() and *RSA_padding_check_PKCS1_OAEP()* were added in OpenSSL 0.9.2b.

NAME

RSA_print, RSA_print_fp, DSAparams_print, DSAparams_print_fp, DSA_print, DSA_print_fp, DHparams_print, DHparams_print_fp − print cryptographic parameters

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_print(BIO *bp, RSA *x, int offset);
int RSA_print_fp(FILE *fp, RSA *x, int offset);

#include <openssl/dsa.h>

int DSAparams_print(BIO *bp, DSA *x);
int DSAparams_print_fp(FILE *fp, DSA *x);
int DSA_print(BIO *bp, DSA *x, int offset);
int DSA_print_fp(FILE *fp, DSA *x, int offset);

#include <openssl/dh.h>

int DHparams_print(BIO *bp, DH *x);
int DHparams_print_fp(FILE *fp, DH *x);
```

DESCRIPTION

A human-readable hexadecimal output of the components of the RSA key, DSA parameters or key or DH parameters is printed to **bp** or **fp**.

The output lines are indented by **offset** spaces.

RETURN VALUES

These functions return 1 on success, 0 on error.

SEE ALSO

openssl_dh (3), *openssl_dsa* (3), *openssl_rsa* (3), *BN_bn2bin* (3)

HISTORY

RSA_print(), *RSA_print_fp()*, *DSA_print()*, *DSA_print_fp()*, *DH_print()*, *DH_print_fp()* are available in all versions of SSLeay and OpenSSL. *DSAparams_print()* and *DSAparams_print_fp()* were added in SSLeay 0.8.

NAME

RSA_private_encrypt, RSA_public_decrypt − low level signature operations

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_private_encrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa, int padding);

int RSA_public_decrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa, int padding);
```

DESCRIPTION

These functions handle RSA signatures at a low level.

RSA_private_encrypt() signs the **flen** bytes at **from** (usually a message digest with an algorithm identifier) using the private key **rsa** and stores the signature in **to**. **to** must point to **RSA_size(rsa)** bytes of memory.

padding denotes one of the following modes:

RSA_PKCS1_PADDING
: PKCS #1 v1.5 padding. This function does not handle the **algorithmIdentifier** specified in PKCS #1. When generating or verifying PKCS #1 signatures, *RSA_sign* (3) and *RSA_verify* (3) should be used.

RSA_NO_PADDING
: Raw RSA signature. This mode should *only* be used to implement cryptographically sound padding modes in the application code. Signing user data directly with RSA is insecure.

RSA_public_decrypt() recovers the message digest from the **flen** bytes long signature at **from** using the signer's public key **rsa**. **to** must point to a memory section large enough to hold the message digest (which is smaller than **RSA_size(rsa) − 11**). **padding** is the padding mode that was used to sign the data.

RETURN VALUES

RSA_private_encrypt() returns the size of the signature (i.e., RSA_size(rsa)). *RSA_public_decrypt()* returns the size of the recovered message digest.

On error, −1 is returned; the error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

ERR_get_error (3), *openssl_rsa* (3), *RSA_sign* (3), *RSA_verify* (3)

HISTORY

The **padding** argument was added in SSLeay 0.8. RSA_NO_PADDING is available since SSLeay 0.9.0.

NAME

RSA_public_encrypt, RSA_private_decrypt − RSA public key cryptography

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_public_encrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa, int padding);

int RSA_private_decrypt(int flen, unsigned char *from,
    unsigned char *to, RSA *rsa, int padding);
```

DESCRIPTION

RSA_public_encrypt() encrypts the **flen** bytes at **from** (usually a session key) using the public key **rsa** and stores the ciphertext in **to**. **to** must point to RSA_size(**rsa**) bytes of memory.

padding denotes one of the following modes:

RSA_PKCS1_PADDING
: PKCS #1 v1.5 padding. This currently is the most widely used mode.

RSA_PKCS1_OAEP_PADDING
: EME-OAEP as defined in PKCS #1 v2.0 with SHA−1, MGF1 and an empty encoding parameter. This mode is recommended for all new applications.

RSA_SSLV23_PADDING
: PKCS #1 v1.5 padding with an SSL-specific modification that denotes that the server is SSL3 capable.

RSA_NO_PADDING
: Raw RSA encryption. This mode should *only* be used to implement cryptographically sound padding modes in the application code. Encrypting user data directly with RSA is insecure.

flen must be less than RSA_size(**rsa**) − 11 for the PKCS #1 v1.5 based padding modes, less than RSA_size(**rsa**) − 41 for RSA_PKCS1_OAEP_PADDING and exactly RSA_size(**rsa**) for RSA_NO_PADDING. The random number generator must be seeded prior to calling *RSA_public_encrypt()*.

RSA_private_decrypt() decrypts the **flen** bytes at **from** using the private key **rsa** and stores the plaintext in **to**. **to** must point to a memory section large enough to hold the decrypted data (which is smaller than RSA_size(**rsa**)). **padding** is the padding mode that was used to encrypt the data.

RETURN VALUES

RSA_public_encrypt() returns the size of the encrypted data (i.e., RSA_size(**rsa**)). *RSA_private_decrypt()* returns the size of the recovered plaintext.

On error, −1 is returned; the error codes can be obtained by *ERR_get_error* (3).

CONFORMING TO

SSL, PKCS #1 v2.0

SEE ALSO

ERR_get_error (3), *openssl_rand* (3), *openssl_rsa* (3), *RSA_size* (3)

HISTORY

The **padding** argument was added in SSLeay 0.8. RSA_NO_PADDING is available since SSLeay 0.9.0, OAEP was added in OpenSSL 0.9.2b.

NAME

RSA_set_default_method, RSA_get_default_method, RSA_set_method, RSA_get_method, RSA_PKCS1_SSLeay, RSA_null_method, RSA_flags, RSA_new_method − select RSA method

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

void RSA_set_default_method(const RSA_METHOD *meth);

RSA_METHOD *RSA_get_default_method(void);

int RSA_set_method(RSA *rsa, const RSA_METHOD *meth);

RSA_METHOD *RSA_get_method(const RSA *rsa);

RSA_METHOD *RSA_PKCS1_SSLeay(void);

RSA_METHOD *RSA_null_method(void);

int RSA_flags(const RSA *rsa);

RSA *RSA_new_method(RSA_METHOD *method);
```

DESCRIPTION

An **RSA_METHOD** specifies the functions that OpenSSL uses for RSA operations. By modifying the method, alternative implementations such as hardware accelerators may be used. IMPORTANT: See the NOTES section for important information about how these RSA API functions are affected by the use of **ENGINE** API calls.

Initially, the default RSA_METHOD is the OpenSSL internal implementation, as returned by *RSA_PKCS1_SSLeay()*.

RSA_set_default_method() makes **meth** the default method for all RSA structures created later. **NB**: This is true only whilst no ENGINE has been set as a default for RSA, so this function is no longer recommended.

RSA_get_default_method() returns a pointer to the current default RSA_METHOD. However, the meaningfulness of this result is dependent on whether the ENGINE API is being used, so this function is no longer recommended.

RSA_set_method() selects **meth** to perform all operations using the key **rsa**. This will replace the RSA_METHOD used by the RSA key and if the previous method was supplied by an ENGINE, the handle to that ENGINE will be released during the change. It is possible to have RSA keys that only work with certain RSA_METHOD implementations (eg. from an ENGINE module that supports embedded hardware-protected keys), and in such cases attempting to change the RSA_METHOD for the key can have unexpected results.

RSA_get_method() returns a pointer to the RSA_METHOD being used by **rsa**. This method may or may not be supplied by an ENGINE implementation, but if it is, the return value can only be guaranteed to be valid as long as the RSA key itself is valid and does not have its implementation changed by *RSA_set_method()*.

RSA_flags() returns the **flags** that are set for **rsa**'s current RSA_METHOD. See the BUGS section.

RSA_new_method() allocates and initializes an RSA structure so that **engine** will be used for the RSA operations. If **engine** is NULL, the default ENGINE for RSA operations is used, and if no default ENGINE is set, the RSA_METHOD controlled by *RSA_set_default_method()* is used.

RSA_flags() returns the **flags** that are set for **rsa**'s current method.

RSA_new_method() allocates and initializes an **RSA** structure so that **method** will be used for the RSA operations. If **method** is **NULL**, the default method is used.

THE RSA_METHOD STRUCTURE

```
typedef struct rsa_meth_st
{
    /* name of the implementation */
       const char *name;

    /* encrypt */
       int (*rsa_pub_enc)(int flen, unsigned char *from,
         unsigned char *to, RSA *rsa, int padding);

    /* verify arbitrary data */
       int (*rsa_pub_dec)(int flen, unsigned char *from,
         unsigned char *to, RSA *rsa, int padding);

    /* sign arbitrary data */
       int (*rsa_priv_enc)(int flen, unsigned char *from,
         unsigned char *to, RSA *rsa, int padding);

    /* decrypt */
       int (*rsa_priv_dec)(int flen, unsigned char *from,
         unsigned char *to, RSA *rsa, int padding);

    /* compute r0 = r0 ^ I mod rsa->n (May be NULL for some
                                       implementations) */
       int (*rsa_mod_exp)(BIGNUM *r0, BIGNUM *I, RSA *rsa);

    /* compute r = a ^ p mod m (May be NULL for some implementations) */
       int (*bn_mod_exp)(BIGNUM *r, BIGNUM *a, const BIGNUM *p,
         const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *m_ctx);

    /* called at RSA_new */
       int (*init)(RSA *rsa);

    /* called at RSA_free */
       int (*finish)(RSA *rsa);

    /* RSA_FLAG_EXT_PKEY        - rsa_mod_exp is called for private key
     *                            operations, even if p,q,dmp1,dmq1,iqmp
     *                            are NULL
     * RSA_FLAG_SIGN_VER        - enable rsa_sign and rsa_verify
     * RSA_METHOD_FLAG_NO_CHECK - don't check pub/private match
     */
       int flags;

       char *app_data; /* ?? */

    /* sign. For backward compatibility, this is used only
     * if (flags & RSA_FLAG_SIGN_VER)
     */
       int (*rsa_sign)(int type, unsigned char *m, unsigned int m_len,
          unsigned char *sigret, unsigned int *siglen, RSA *rsa);

    /* verify. For backward compatibility, this is used only
     * if (flags & RSA_FLAG_SIGN_VER)
     */
       int (*rsa_verify)(int type, unsigned char *m, unsigned int m_len,
          unsigned char *sigbuf, unsigned int siglen, RSA *rsa);

} RSA_METHOD;
```

RETURN VALUES

RSA_PKCS1_SSLeay(), *RSA_PKCS1_null_method()*, *RSA_get_default_method()* and *RSA_get_method()* return pointers to the respective RSA_METHODs.

RSA_set_default_method() returns no value.

RSA_set_method() returns a pointer to the old RSA_METHOD implementation that was replaced. However, this return value should probably be ignored because if it was supplied by an ENGINE, the pointer could be invalidated at any time if the ENGINE is unloaded (in fact it could be unloaded as a result of the *RSA_set_method()* function releasing its handle to the ENGINE). For this reason, the return type may be replaced with a **void** declaration in a future release.

RSA_new_method() returns NULL and sets an error code that can be obtained by *ERR_get_error* (3) if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

NOTES

As of version 0.9.7, RSA_METHOD implementations are grouped together with other algorithmic APIs (eg. DSA_METHOD, EVP_CIPHER, etc) into **ENGINE** modules. If a default ENGINE is specified for RSA functionality using an ENGINE API function, that will override any RSA defaults set using the RSA API (ie. *RSA_set_default_method()*). For this reason, the ENGINE API is the recommended way to control default implementations for use in RSA and other cryptographic algorithms.

BUGS

The behaviour of *RSA_flags()* is a mis-feature that is left as-is for now to avoid creating compatibility problems. RSA functionality, such as the encryption functions, are controlled by the **flags** value in the RSA key itself, not by the **flags** value in the RSA_METHOD attached to the RSA key (which is what this function returns). If the flags element of an RSA key is changed, the changes will be honoured by RSA functionality but will not be reflected in the return value of the *RSA_flags()* function − in effect *RSA_flags()* behaves more like an *RSA_default_flags()* function (which does not currently exist).

SEE ALSO

openssl_rsa (3), *RSA_new* (3)

HISTORY

RSA_new_method() and *RSA_set_default_method()* appeared in SSLeay 0.8. *RSA_get_default_method()*, *RSA_set_method()* and *RSA_get_method()* as well as the rsa_sign and rsa_verify components of RSA_METHOD were added in OpenSSL 0.9.4.

RSA_set_default_openssl_method() and *RSA_get_default_openssl_method()* replaced *RSA_set_default_method()* and *RSA_get_default_method()* respectively, and *RSA_set_method()* and *RSA_new_method()* were altered to use **ENGINE**s rather than **RSA_METHOD**s during development of the engine version of OpenSSL 0.9.6. For 0.9.7, the handling of defaults in the ENGINE API was restructured so that this change was reversed, and behaviour of the other functions resembled more closely the previous behaviour. The behaviour of defaults in the ENGINE API now transparently overrides the behaviour of defaults in the RSA API without requiring changing these function prototypes.

NAME

RSA_sign, RSA_verify − RSA signatures

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_sign(int type, const unsigned char *m, unsigned int m_len,
   unsigned char *sigret, unsigned int *siglen, RSA *rsa);

int RSA_verify(int type, const unsigned char *m, unsigned int m_len,
   unsigned char *sigbuf, unsigned int siglen, RSA *rsa);
```

DESCRIPTION

RSA_sign() signs the message digest **m** of size **m_len** using the private key **rsa** as specified in PKCS #1 v2.0. It stores the signature in **sigret** and the signature size in **siglen**. **sigret** must point to RSA_size(**rsa**) bytes of memory.

type denotes the message digest algorithm that was used to generate **m**. It usually is one of **NID_sha1**, **NID_ripemd160** and **NID_md5**; see *objects*(3) for details. If **type** is **NID_md5_sha1**, an SSL signature (MD5 and SHA1 message digests with PKCS #1 padding and no algorithm identifier) is created.

RSA_verify() verifies that the signature **sigbuf** of size **siglen** matches a given message digest **m** of size **m_len**. **type** denotes the message digest algorithm that was used to generate the signature. **rsa** is the signer's public key.

RETURN VALUES

RSA_sign() returns 1 on success, 0 otherwise. *RSA_verify()* returns 1 on successful verification, 0 otherwise.

The error codes can be obtained by *ERR_get_error*(3).

BUGS

Certain signatures with an improper algorithm identifier are accepted for compatibility with SSLeay 0.4.5 :−)

CONFORMING TO

SSL, PKCS #1 v2.0

SEE ALSO

ERR_get_error(3), *objects*(3), *openssl_rsa*(3), *RSA_private_encrypt*(3), *RSA_public_decrypt*(3)

HISTORY

RSA_sign() and *RSA_verify()* are available in all versions of SSLeay and OpenSSL.

NAME

RSA_sign_ASN1_OCTET_STRING, RSA_verify_ASN1_OCTET_STRING − RSA signatures

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_sign_ASN1_OCTET_STRING(int dummy, unsigned char *m,
   unsigned int m_len, unsigned char *sigret, unsigned int *siglen,
   RSA *rsa);

int RSA_verify_ASN1_OCTET_STRING(int dummy, unsigned char *m,
   unsigned int m_len, unsigned char *sigbuf, unsigned int siglen,
   RSA *rsa);
```

DESCRIPTION

RSA_sign_ASN1_OCTET_STRING() signs the octet string **m** of size **m_len** using the private key **rsa** represented in DER using PKCS #1 padding. It stores the signature in **sigret** and the signature size in **siglen**. **sigret** must point to **RSA_size(rsa)** bytes of memory.

dummy is ignored.

The random number generator must be seeded prior to calling *RSA_sign_ASN1_OCTET_STRING()*.

RSA_verify_ASN1_OCTET_STRING() verifies that the signature **sigbuf** of size **siglen** is the DER representation of a given octet string **m** of size **m_len**. **dummy** is ignored. **rsa** is the signer's public key.

RETURN VALUES

RSA_sign_ASN1_OCTET_STRING() returns 1 on success, 0 otherwise. *RSA_verify_ASN1_OCTET_STRING()* returns 1 on successful verification, 0 otherwise.

The error codes can be obtained by *ERR_get_error* (3).

BUGS

These functions serve no recognizable purpose.

SEE ALSO

ERR_get_error (3), *objects* (3), *openssl_rand* (3), *openssl_rsa* (3), *RSA_sign* (3), *RSA_verify* (3)

HISTORY

RSA_sign_ASN1_OCTET_STRING() and *RSA_verify_ASN1_OCTET_STRING()* were added in SSLeay 0.8.

NAME

RSA_size − get RSA modulus size

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_size(const RSA *rsa);
```

DESCRIPTION

This function returns the RSA modulus size in bytes. It can be used to determine how much memory must be allocated for an RSA encrypted value.

rsa->n must not be **NULL**.

RETURN VALUE

The size in bytes.

SEE ALSO

openssl_rsa (3)

HISTORY

RSA_size() is available in all versions of SSLeay and OpenSSL.

NAME

SMIME_read_PKCS7 − parse S/MIME message.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/pkcs7.h>

PKCS7 *SMIME_read_PKCS7(BIO *in, BIO **bcont);
```

DESCRIPTION

SMIME_read_PKCS7() parses a message in S/MIME format.

in is a BIO to read the message from.

If cleartext signing is used then the content is saved in a memory bio which is written to ***bcont**, otherwise ***bcont** is set to **NULL**.

The parsed PKCS#7 structure is returned or **NULL** if an error occurred.

NOTES

If ***bcont** is not **NULL** then the message is clear text signed. ***bcont** can then be passed to *PKCS7_verify()* with the **PKCS7_DETACHED** flag set.

Otherwise the type of the returned structure can be determined using *PKCS7_type()*.

To support future functionality if **bcont** is not **NULL** ***bcont** should be initialized to **NULL**. For example:

```
BIO *cont = NULL;
PKCS7 *p7;

p7 = SMIME_read_PKCS7(in, &cont);
```

BUGS

The MIME parser used by *SMIME_read_PKCS7()* is somewhat primitive. While it will handle most S/MIME messages more complex compound formats may not work.

The parser assumes that the PKCS7 structure is always base64 encoded and will not handle the case where it is in binary format or uses quoted printable format.

The use of a memory BIO to hold the signed content limits the size of message which can be processed due to memory restraints: a streaming single pass option should be available.

RETURN VALUES

SMIME_read_PKCS7() returns a valid **PKCS7** structure or **NULL** is an error occurred. The error can be obtained from *ERR_get_error* (3).

SEE ALSO

ERR_get_error (3), *PKCS7_type* (3) *SMIME_read_PKCS7* (3), *PKCS7_sign* (3), *PKCS7_verify* (3), *PKCS7_encrypt* (3) *PKCS7_decrypt* (3)

HISTORY

SMIME_read_PKCS7() was added to OpenSSL 0.9.5

NAME

SMIME_write_PKCS7 − convert PKCS#7 structure to S/MIME format.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/pkcs7.h>

int SMIME_write_PKCS7(BIO *out, PKCS7 *p7, BIO *data, int flags);
```

DESCRIPTION

SMIME_write_PKCS7() adds the appropriate MIME headers to a PKCS#7 structure to produce an S/MIME message.

out is the BIO to write the data to. **p7** is the appropriate **PKCS7** structure. If streaming is enabled then the content must be supplied in the **data** argument. **flags** is an optional set of flags.

NOTES

The following flags can be passed in the **flags** parameter.

If **PKCS7_DETACHED** is set then cleartext signing will be used, this option only makes sense for signedData where **PKCS7_DETACHED** is also set when *PKCS7_sign()* is also called.

If the **PKCS7_TEXT** flag is set MIME headers for type **text/plain** are added to the content, this only makes sense if **PKCS7_DETACHED** is also set.

If the **PKCS7_STREAM** flag is set streaming is performed. This flag should only be set if **PKCS7_STREAM** was also set in the previous call to *PKCS7_sign()* or ***PKCS7_encrypt()***.

If cleartext signing is being used and **PKCS7_STREAM** not set then the data must be read twice: once to compute the signature in *PKCS7_sign()* and once to output the S/MIME message.

If streaming is performed the content is output in BER format using indefinite length constructuted encoding except in the case of signed data with detached content where the content is absent and DER format is used.

BUGS

SMIME_write_PKCS7() always base64 encodes PKCS#7 structures, there should be an option to disable this.

RETURN VALUES

SMIME_write_PKCS7() returns 1 for success or 0 for failure.

SEE ALSO

ERR_get_error (3), *PKCS7_sign* (3), *PKCS7_verify* (3), *PKCS7_encrypt* (3) *PKCS7_decrypt* (3)

HISTORY

SMIME_write_PKCS7() was added to OpenSSL 0.9.5

NAME

SSL_CIPHER_get_name, SSL_CIPHER_get_bits, SSL_CIPHER_get_version, SSL_CIPHER_description − get SSL_CIPHER properties

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

const char *SSL_CIPHER_get_name(const SSL_CIPHER *cipher);
int SSL_CIPHER_get_bits(const SSL_CIPHER *cipher, int *alg_bits);
char *SSL_CIPHER_get_version(const SSL_CIPHER *cipher);
char *SSL_CIPHER_description(SSL_CIPHER *cipher, char *buf, int size);
```

DESCRIPTION

SSL_CIPHER_get_name() returns a pointer to the name of **cipher**. If the argument is the NULL pointer, a pointer to the constant value ''NONE'' is returned.

SSL_CIPHER_get_bits() returns the number of secret bits used for **cipher**. If **alg_bits** is not NULL, it contains the number of bits processed by the chosen algorithm. If **cipher** is NULL, 0 is returned.

SSL_CIPHER_get_version() returns the protocol version for **cipher**, currently ''SSLv2'', ''SSLv3'', or ''TLSv1''. If **cipher** is NULL, ''(NONE)'' is returned.

SSL_CIPHER_description() returns a textual description of the cipher used into the buffer **buf** of length **len** provided. **len** must be at least 128 bytes, otherwise a pointer to the the string ''Buffer too small'' is returned. If **buf** is NULL, a buffer of 128 bytes is allocated using *OPENSSL_malloc()*. If the allocation fails, a pointer to the string ''OPENSSL_malloc Error'' is returned.

NOTES

The number of bits processed can be different from the secret bits. An export cipher like e.g. EXP−RC4−MD5 has only 40 secret bits. The algorithm does use the full 128 bits (which would be returned for **alg_bits**), of which however 88bits are fixed. The search space is hence only 40 bits.

The string returned by *SSL_CIPHER_description()* in case of success consists of cleartext information separated by one or more blanks in the following sequence:

<ciphername>
: Textual representation of the cipher name.

<protocol version>
: Protocol version: **SSLv2**, **SSLv3**. The TLSv1 ciphers are flagged with SSLv3.

Kx=<key exchange>
: Key exchange method: **RSA** (for export ciphers as **RSA(512)** or **RSA(1024)**), **DH** (for export ciphers as **DH(512)** or **DH(1024)**), **DH/RSA**, **DH/DSS**, **Fortezza**.

Au=<authentication>
: Authentication method: **RSA**, **DSS**, **DH**, **None**. None is the representation of anonymous ciphers.

Enc=<symmetric encryption method>
: Encryption method with number of secret bits: **DES(40)**, **DES(56)**, **3DES(168)**, **RC4(40)**, **RC4(56)**, **RC4(64)**, **RC4(128)**, **RC2(40)**, **RC2(56)**, **RC2(128)**, **IDEA(128)**, **Fortezza**, **None**.

Mac=<message authentication code>
: Message digest: **MD5**, **SHA1**.

<export flag>
: If the cipher is flagged exportable with respect to old US crypto regulations, the word "**export**" is printed.

EXAMPLES

Some examples for the output of *SSL_CIPHER_description()*:

```
EDH-RSA-DES-CBC3-SHA    SSLv3 Kx=DH       Au=RSA  Enc=3DES(168) Mac=SHA1
EDH-DSS-DES-CBC3-SHA    SSLv3 Kx=DH       Au=DSS  Enc=3DES(168) Mac=SHA1
RC4-MD5                 SSLv3 Kx=RSA      Au=RSA  Enc=RC4(128)  Mac=MD5
EXP-RC4-MD5             SSLv3 Kx=RSA(512) Au=RSA  Enc=RC4(40)   Mac=MD5  export
```

BUGS

If *SSL_CIPHER_description()* is called with **cipher** being NULL, the library crashes.

If *SSL_CIPHER_description()* cannot handle a built-in cipher, the according description of the cipher property is **unknown**. This case should not occur.

RETURN VALUES

See DESCRIPTION

SEE ALSO

ssl (3), *SSL_get_current_cipher* (3), *SSL_get_ciphers* (3), *openssl_ciphers* (1)

NAME

SSL_COMP_add_compression_method − handle SSL/TLS integrated compression methods

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_COMP_add_compression_method(int id, COMP_METHOD *cm);
```

DESCRIPTION

SSL_COMP_add_compression_method() adds the compression method **cm** with the identifier **id** to the list of available compression methods. This list is globally maintained for all SSL operations within this application. It cannot be set for specific SSL_CTX or SSL objects.

NOTES

The TLS standard (or SSLv3) allows the integration of compression methods into the communication. The TLS RFC does however not specify compression methods or their corresponding identifiers, so there is currently no compatible way to integrate compression with unknown peers. It is therefore currently not recommended to integrate compression into applications. Applications for non-public use may agree on certain compression methods. Using different compression methods with the same identifier will lead to connection failure.

An OpenSSL client speaking a protocol that allows compression (SSLv3, TLSv1) will unconditionally send the list of all compression methods enabled with *SSL_COMP_add_compression_method()* to the server during the handshake. Unlike the mechanisms to set a cipher list, there is no method available to restrict the list of compression method on a per connection basis.

An OpenSSL server will match the identifiers listed by a client against its own compression methods and will unconditionally activate compression when a matching identifier is found. There is no way to restrict the list of compression methods supported on a per connection basis.

The OpenSSL library has the compression methods ***COMP_rle()*** and (when especially enabled during compilation) ***COMP_zlib()*** available.

WARNINGS

Once the identities of the compression methods for the TLS protocol have been standardized, the compression API will most likely be changed. Using it in the current state is not recommended.

RETURN VALUES

SSL_COMP_add_compression_method() may return the following values:

0 The operation succeeded.

1 The operation failed. Check the error queue to find out the reason.

SEE ALSO

ssl (3)

NAME

SSL_CTX_add_extra_chain_cert − add certificate to chain

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_CTX_add_extra_chain_cert(SSL_CTX ctx, X509 *x509)
```

DESCRIPTION

SSL_CTX_add_extra_chain_cert() adds the certificate **x509** to the certificate chain presented together with the certificate. Several certificates can be added one after the other.

NOTES

When constructing the certificate chain, the chain will be formed from these certificates explicitly specified. If no chain is specified, the library will try to complete the chain from the available CA certificates in the trusted CA storage, see *SSL_CTX_load_verify_locations* (3).

RETURN VALUES

SSL_CTX_add_extra_chain_cert() returns 1 on success. Check out the error stack to find out the reason for failure otherwise.

SEE ALSO

ssl (3), *SSL_CTX_use_certificate* (3), *SSL_CTX_set_client_cert_cb* (3), *SSL_CTX_load_verify_locations* (3)

NAME

SSL_CTX_add_session, SSL_add_session, SSL_CTX_remove_session, SSL_remove_session − manipulate session cache

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_CTX_add_session(SSL_CTX *ctx, SSL_SESSION *c);
int SSL_add_session(SSL_CTX *ctx, SSL_SESSION *c);

int SSL_CTX_remove_session(SSL_CTX *ctx, SSL_SESSION *c);
int SSL_remove_session(SSL_CTX *ctx, SSL_SESSION *c);
```

DESCRIPTION

SSL_CTX_add_session() adds the session **c** to the context **ctx**. The reference count for session **c** is incremented by 1. If a session with the same session id already exists, the old session is removed by calling *SSL_SESSION_free* (3).

SSL_CTX_remove_session() removes the session **c** from the context **ctx**. *SSL_SESSION_free* (3) is called once for **c**.

SSL_add_session() and *SSL_remove_session()* are synonyms for their SSL_CTX_*() counterparts.

NOTES

When adding a new session to the internal session cache, it is examined whether a session with the same session id already exists. In this case it is assumed that both sessions are identical. If the same session is stored in a different SSL_SESSION object, The old session is removed and replaced by the new session. If the session is actually identical (the SSL_SESSION object is identical), *SSL_CTX_add_session()* is a no−op, and the return value is 0.

If a server SSL_CTX is configured with the SSL_SESS_CACHE_NO_INTERNAL_STORE flag then the internal cache will not be populated automatically by new sessions negotiated by the SSL/TLS implementation, even though the internal cache will be searched automatically for session-resume requests (the latter can be surpressed by SSL_SESS_CACHE_NO_INTERNAL_LOOKUP). So the application can use *SSL_CTX_add_session()* directly to have full control over the sessions that can be resumed if desired.

RETURN VALUES

The following values are returned by all functions:

0

```
The operation failed. In case of the add operation, it was tried to add
the same (identical) session twice. In case of the remove operation, the
session was not found in the cache.
```

1

```
The operation succeeded.
```

SEE ALSO

ssl (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_SESSION_free* (3)

NAME

SSL_CTX_ctrl, SSL_CTX_callback_ctrl, SSL_ctrl, SSL_callback_ctrl − internal handling functions for SSL_CTX and SSL objects

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_CTX_ctrl(SSL_CTX *ctx, int cmd, long larg, void *parg);
long SSL_CTX_callback_ctrl(SSL_CTX *, int cmd, void (*fp)());

long SSL_ctrl(SSL *ssl, int cmd, long larg, void *parg);
long SSL_callback_ctrl(SSL *, int cmd, void (*fp)());
```

DESCRIPTION

The SSL_*_*ctrl()* family of functions is used to manipulate settings of the SSL_CTX and SSL objects. Depending on the command **cmd** the arguments **larg**, **parg**, or **fp** are evaluated. These functions should never be called directly. All functionalities needed are made available via other functions or macros.

RETURN VALUES

The return values of the SSL*_*ctrl()* functions depend on the command supplied via the **cmd** parameter.

SEE ALSO

ssl (3)

NAME

SSL_CTX_flush_sessions, SSL_flush_sessions − remove expired sessions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_flush_sessions(SSL_CTX *ctx, long tm);
void SSL_flush_sessions(SSL_CTX *ctx, long tm);
```

DESCRIPTION

SSL_CTX_flush_sessions() causes a run through the session cache of **ctx** to remove sessions expired at time **tm**.

SSL_flush_sessions() is a synonym for *SSL_CTX_flush_sessions()*.

NOTES

If enabled, the internal session cache will collect all sessions established up to the specified maximum number (see *SSL_CTX_sess_set_cache_size()*). As sessions will not be reused ones they are expired, they should be removed from the cache to save resources. This can either be done automatically whenever 255 new sessions were established (see *SSL_CTX_set_session_cache_mode*(3)) or manually by calling *SSL_CTX_flush_sessions()*.

The parameter **tm** specifies the time which should be used for the expiration test, in most cases the actual time given by *time*(0) will be used.

SSL_CTX_flush_sessions() will only check sessions stored in the internal cache. When a session is found and removed, the remove_session_cb is however called to synchronize with the external cache (see *SSL_CTX_sess_set_get_cb*(3)).

RETURN VALUES

SEE ALSO

ssl(3), *SSL_CTX_set_session_cache_mode*(3), *SSL_CTX_set_timeout*(3), *SSL_CTX_sess_set_get_cb*(3)

NAME

SSL_CTX_free − free an allocated SSL_CTX object

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_free(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_free() decrements the reference count of **ctx**, and removes the SSL_CTX object pointed to by **ctx** and frees up the allocated memory if the the reference count has reached 0.

It also calls the *free()*ing procedures for indirectly affected items, if applicable: the session cache, the list of ciphers, the list of Client CAs, the certificates and keys.

WARNINGS

If a session-remove callback is set (*SSL_CTX_sess_set_remove_cb()*), this callback will be called for each session being freed from **ctx**'s session cache. This implies, that all corresponding sessions from an external session cache are removed as well. If this is not desired, the user should explicitly unset the callback by calling SSL_CTX_sess_set_remove_cb(**ctx**, NULL) prior to calling *SSL_CTX_free()*.

RETURN VALUES

SSL_CTX_free() does not provide diagnostic information.

SEE ALSO

SSL_CTX_new (3), *ssl* (3), *SSL_CTX_sess_set_get_cb* (3)

NAME

SSL_CTX_get_ex_new_index, SSL_CTX_set_ex_data, SSL_CTX_get_ex_data − internal application specific data functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_CTX_get_ex_new_index(long argl, void *argp,
                CRYPTO_EX_new *new_func,
                CRYPTO_EX_dup *dup_func,
                CRYPTO_EX_free *free_func);

int SSL_CTX_set_ex_data(SSL_CTX *ctx, int idx, void *arg);

void *SSL_CTX_get_ex_data(const SSL_CTX *ctx, int idx);

typedef int new_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
                int idx, long argl, void *argp);
typedef void free_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
                int idx, long argl, void *argp);
typedef int dup_func(CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d,
                int idx, long argl, void *argp);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

SSL_CTX_get_ex_new_index() is used to register a new index for application specific data.

SSL_CTX_set_ex_data() is used to store application data at **arg** for **idx** into the **ctx** object.

SSL_CTX_get_ex_data() is used to retrieve the information for **idx** from **ctx**.

A detailed description for the ****_get_ex_new_index()*** functionality can be found in *RSA_get_ex_new_index* (3). The ****_get_ex_data()*** and ****_set_ex_data()*** functionality is described in *CRYPTO_set_ex_data* (3).

SEE ALSO

ssl (3), *RSA_get_ex_new_index* (3), *CRYPTO_set_ex_data* (3)

NAME

SSL_CTX_get_verify_mode, SSL_get_verify_mode, SSL_CTX_get_verify_depth, SSL_get_verify_depth, SSL_get_verify_callback, SSL_CTX_get_verify_callback − get currently set verification parameters

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_CTX_get_verify_mode(const SSL_CTX *ctx);
int SSL_get_verify_mode(const SSL *ssl);
int SSL_CTX_get_verify_depth(const SSL_CTX *ctx);
int SSL_get_verify_depth(const SSL *ssl);
int (*SSL_CTX_get_verify_callback(const SSL_CTX *ctx))(int, X509_STORE_CTX *);
int (*SSL_get_verify_callback(const SSL *ssl))(int, X509_STORE_CTX *);
```

DESCRIPTION

SSL_CTX_get_verify_mode() returns the verification mode currently set in **ctx**.

SSL_get_verify_mode() returns the verification mode currently set in **ssl**.

SSL_CTX_get_verify_depth() returns the verification depth limit currently set in **ctx**. If no limit has been explicitly set, −1 is returned and the default value will be used.

SSL_get_verify_depth() returns the verification depth limit currently set in **ssl**. If no limit has been explicitly set, −1 is returned and the default value will be used.

SSL_CTX_get_verify_callback() returns a function pointer to the verification callback currently set in **ctx**. If no callback was explicitly set, the NULL pointer is returned and the default callback will be used.

SSL_get_verify_callback() returns a function pointer to the verification callback currently set in **ssl**. If no callback was explicitly set, the NULL pointer is returned and the default callback will be used.

RETURN VALUES

See DESCRIPTION

SEE ALSO

ssl (3), *SSL_CTX_set_verify* (3)

NAME

SSL_CTX_load_verify_locations − set default locations for trusted CA certificates

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_CTX_load_verify_locations(SSL_CTX *ctx, const char *CAfile,
                                  const char *CApath);
```

DESCRIPTION

SSL_CTX_load_verify_locations() specifies the locations for **ctx**, at which CA certificates for verification purposes are located. The certificates available via **CAfile** and **CApath** are trusted.

NOTES

If **CAfile** is not NULL, it points to a file of CA certificates in PEM format. The file can contain several CA certificates identified by

```
-----BEGIN CERTIFICATE-----
... (CA certificate in base64 encoding) ...
-----END CERTIFICATE-----
```

sequences. Before, between, and after the certificates text is allowed which can be used e.g. for descriptions of the certificates.

The **CAfile** is processed on execution of the *SSL_CTX_load_verify_locations()* function.

If **CApath** is not NULL, it points to a directory containing CA certificates in PEM format. The files each contain one CA certificate. The files are looked up by the CA subject name hash value, which must hence be available. If more than one CA certificate with the same name hash value exist, the extension must be different (e.g. 9d66eef0.0, 9d66eef0.1 etc). The search is performed in the ordering of the extension number, regardless of other properties of the certificates. Use the **c_rehash** utility to create the necessary links.

The certificates in **CApath** are only looked up when required, e.g. when building the certificate chain or when actually performing the verification of a peer certificate.

When looking up CA certificates, the OpenSSL library will first search the certificates in **CAfile**, then those in **CApath**. Certificate matching is done based on the subject name, the key identifier (if present), and the serial number as taken from the certificate to be verified. If these data do not match, the next certificate will be tried. If a first certificate matching the parameters is found, the verification process will be performed; no other certificates for the same parameters will be searched in case of failure.

In server mode, when requesting a client certificate, the server must send the list of CAs of which it will accept client certificates. This list is not influenced by the contents of **CAfile** or **CApath** and must explicitly be set using the *SSL_CTX_set_client_CA_list* (3) family of functions.

When building its own certificate chain, an OpenSSL client/server will try to fill in missing certificates from **CAfile**/**CApath**, if the certificate chain was not explicitly specified (see *SSL_CTX_add_extra_chain_cert* (3), *SSL_CTX_use_certificate* (3).

WARNINGS

If several CA certificates matching the name, key identifier, and serial number condition are available, only the first one will be examined. This may lead to unexpected results if the same CA certificate is available with different expiration dates. If a ‘‘certificate expired’’ verification error occurs, no other certificate will be searched. Make sure to not have expired certificates mixed with valid ones.

EXAMPLES

Generate a CA certificate file with descriptive text from the CA certificates ca1.pem ca2.pem ca3.pem:

```
#!/bin/sh
rm CAfile.pem
for i in ca1.pem ca2.pem ca3.pem ; do
  openssl x509 -in $i -text >> CAfile.pem
done
```

Prepare the directory /some/where/certs containing several CA certificates for use as **CApath**:

```
cd /some/where/certs
c_rehash .
```

RETURN VALUES

The following return values can occur:

0 The operation failed because **CAfile** and **CApath** are NULL or the processing at one of the locations specified failed. Check the error stack to find out the reason.

1 The operation succeeded.

SEE ALSO

ssl (3), *SSL_CTX_set_client_CA_list* (3), *SSL_get_client_CA_list* (3), *SSL_CTX_use_certificate* (3), *SSL_CTX_add_extra_chain_cert* (3), *SSL_CTX_set_cert_store* (3)

NAME

SSL_CTX_new − create a new SSL_CTX object as framework for TLS/SSL enabled functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

SSL_CTX *SSL_CTX_new(const SSL_METHOD *method);
```

DESCRIPTION

SSL_CTX_new() creates a new **SSL_CTX** object as framework to establish TLS/SSL enabled connections.

NOTES

The SSL_CTX object uses **method** as connection method. The methods exist in a generic type (for client and server use), a server only type, and a client only type. **method** can be of the following types:

SSLv2_method(void), SSLv2_server_method(void), SSLv2_client_method(void)
: A TLS/SSL connection established with these methods will only understand the SSLv2 protocol. A client will send out SSLv2 client hello messages and will also indicate that it only understand SSLv2. A server will only understand SSLv2 client hello messages.

SSLv3_method(void), SSLv3_server_method(void), SSLv3_client_method(void)
: A TLS/SSL connection established with these methods will only understand the SSLv3 protocol. A client will send out SSLv3 client hello messages and will indicate that it only understands SSLv3. A server will only understand SSLv3 client hello messages. This especially means, that it will not understand SSLv2 client hello messages which are widely used for compatibility reasons, see SSLv23_*_*method()*.

TLSv1_method(void), TLSv1_server_method(void), TLSv1_client_method(void)
: A TLS/SSL connection established with these methods will only understand the TLSv1 protocol. A client will send out TLSv1 client hello messages and will indicate that it only understands TLSv1. A server will only understand TLSv1 client hello messages. This especially means, that it will not understand SSLv2 client hello messages which are widely used for compatibility reasons, see SSLv23_*_*method()*. It will also not understand SSLv3 client hello messages.

SSLv23_method(void), SSLv23_server_method(void), SSLv23_client_method(void)
: A TLS/SSL connection established with these methods will understand the SSLv2, SSLv3, and TLSv1 protocol. A client will send out SSLv2 client hello messages and will indicate that it also understands SSLv3 and TLSv1. A server will understand SSLv2, SSLv3, and TLSv1 client hello messages. This is the best choice when compatibility is a concern.

The list of protocols available can later be limited using the SSL_OP_NO_SSLv2, SSL_OP_NO_SSLv3, SSL_OP_NO_TLSv1 options of the ***SSL_CTX_set_options()*** or ***SSL_set_options()*** functions. Using these options it is possible to choose e.g. *SSLv23_server_method()* and be able to negotiate with all possible clients, but to only allow newer protocols like SSLv3 or TLSv1.

SSL_CTX_new() initializes the list of ciphers, the session cache setting, the callbacks, the keys and certificates, and the options to its default values.

RETURN VALUES

The following return values can occur:

NULL
: The creation of a new SSL_CTX object failed. Check the error stack to find out the reason.

Pointer to an SSL_CTX object
: The return value points to an allocated SSL_CTX object.

SEE ALSO

SSL_CTX_free (3), *SSL_accept* (3), *ssl* (3), *SSL_set_connect_state* (3)

NAME

SSL_CTX_sess_number, SSL_CTX_sess_connect, SSL_CTX_sess_connect_good, SSL_CTX_sess_connect_renegotiate, SSL_CTX_sess_accept, SSL_CTX_sess_accept_good, SSL_CTX_sess_accept_renegotiate, SSL_CTX_sess_hits, SSL_CTX_sess_cb_hits, SSL_CTX_sess_misses, SSL_CTX_sess_timeouts, SSL_CTX_sess_cache_full − obtain session cache statistics

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_CTX_sess_number(SSL_CTX *ctx);
long SSL_CTX_sess_connect(SSL_CTX *ctx);
long SSL_CTX_sess_connect_good(SSL_CTX *ctx);
long SSL_CTX_sess_connect_renegotiate(SSL_CTX *ctx);
long SSL_CTX_sess_accept(SSL_CTX *ctx);
long SSL_CTX_sess_accept_good(SSL_CTX *ctx);
long SSL_CTX_sess_accept_renegotiate(SSL_CTX *ctx);
long SSL_CTX_sess_hits(SSL_CTX *ctx);
long SSL_CTX_sess_cb_hits(SSL_CTX *ctx);
long SSL_CTX_sess_misses(SSL_CTX *ctx);
long SSL_CTX_sess_timeouts(SSL_CTX *ctx);
long SSL_CTX_sess_cache_full(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_sess_number() returns the current number of sessions in the internal session cache.

SSL_CTX_sess_connect() returns the number of started SSL/TLS handshakes in client mode.

SSL_CTX_sess_connect_good() returns the number of successfully established SSL/TLS sessions in client mode.

SSL_CTX_sess_connect_renegotiate() returns the number of start renegotiations in client mode.

SSL_CTX_sess_accept() returns the number of started SSL/TLS handshakes in server mode.

SSL_CTX_sess_accept_good() returns the number of successfully established SSL/TLS sessions in server mode.

SSL_CTX_sess_accept_renegotiate() returns the number of start renegotiations in server mode.

SSL_CTX_sess_hits() returns the number of successfully reused sessions. In client mode a session set with *SSL_set_session* (3) successfully reused is counted as a hit. In server mode a session successfully retrieved from internal or external cache is counted as a hit.

SSL_CTX_sess_cb_hits() returns the number of successfully retrieved sessions from the external session cache in server mode.

SSL_CTX_sess_misses() returns the number of sessions proposed by clients that were not found in the internal session cache in server mode.

SSL_CTX_sess_timeouts() returns the number of sessions proposed by clients and either found in the internal or external session cache in server mode,
but that were invalid due to timeout. These sessions are not included in the *SSL_CTX_sess_hits()* count.

SSL_CTX_sess_cache_full() returns the number of sessions that were removed because the maximum session cache size was exceeded.

RETURN VALUES

The functions return the values indicated in the DESCRIPTION section.

SEE ALSO

ssl (3), *SSL_set_session* (3), *SSL_CTX_set_session_cache_mode* (3) *SSL_CTX_sess_set_cache_size* (3)

NAME

SSL_CTX_sess_set_cache_size, SSL_CTX_sess_get_cache_size − manipulate session cache size

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_CTX_sess_set_cache_size(SSL_CTX *ctx, long t);
long SSL_CTX_sess_get_cache_size(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_sess_set_cache_size() sets the size of the internal session cache of context **ctx** to **t**.

SSL_CTX_sess_get_cache_size() returns the currently valid session cache size.

NOTES

The internal session cache size is SSL_SESSION_CACHE_MAX_SIZE_DEFAULT, currently 1024*20, so that up to 20000 sessions can be held. This size can be modified using the *SSL_CTX_sess_set_cache_size()* call. A special case is the size 0, which is used for unlimited size.

When the maximum number of sessions is reached, no more new sessions are added to the cache. New space may be added by calling *SSL_CTX_flush_sessions* (3) to remove expired sessions.

If the size of the session cache is reduced and more sessions are already in the session cache, old session will be removed at the next time a session shall be added. This removal is not synchronized with the expiration of sessions.

RETURN VALUES

SSL_CTX_sess_set_cache_size() returns the previously valid size.

SSL_CTX_sess_get_cache_size() returns the currently valid size.

SEE ALSO

ssl (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_CTX_sess_number* (3), *SSL_CTX_flush_sessions* (3)

NAME

SSL_CTX_sess_set_new_cb, SSL_CTX_sess_set_remove_cb, SSL_CTX_sess_set_get_cb, SSL_CTX_sess_get_new_cb, SSL_CTX_sess_get_remove_cb, SSL_CTX_sess_get_get_cb − provide callback functions for server side external session caching

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_sess_set_new_cb(SSL_CTX *ctx,
                             int (*new_session_cb)(SSL *, SSL_SESSION *));
void SSL_CTX_sess_set_remove_cb(SSL_CTX *ctx,
          void (*remove_session_cb)(SSL_CTX *ctx, SSL_SESSION *));
void SSL_CTX_sess_set_get_cb(SSL_CTX *ctx,
          SSL_SESSION (*get_session_cb)(SSL *, unsigned char *, int, int *));

int (*SSL_CTX_sess_get_new_cb(SSL_CTX *ctx))(struct ssl_st *ssl, SSL_SESSION *sess)
void (*SSL_CTX_sess_get_remove_cb(SSL_CTX *ctx))(struct ssl_ctx_st *ctx, SSL_SESSIO
SSL_SESSION *(*SSL_CTX_sess_get_get_cb(SSL_CTX *ctx))(struct ssl_st *ssl, unsigned

int (*new_session_cb)(struct ssl_st *ssl, SSL_SESSION *sess);
void (*remove_session_cb)(struct ssl_ctx_st *ctx, SSL_SESSION *sess);
SSL_SESSION *(*get_session_cb)(struct ssl_st *ssl, unsigned char *data,
             int len, int *copy);
```

DESCRIPTION

SSL_CTX_sess_set_new_cb() sets the callback function, which is automatically called whenever a new session was negotiated.

SSL_CTX_sess_set_remove_cb() sets the callback function, which is automatically called whenever a session is removed by the SSL engine, because it is considered faulty or the session has become obsolete because of exceeding the timeout value.

SSL_CTX_sess_set_get_cb() sets the callback function which is called, whenever a SSL/TLS client proposed to resume a session but the session could not be found in the internal session cache (see *SSL_CTX_set_session_cache_mode* (3)). (SSL/TLS server only.)

SSL_CTX_sess_get_new_cb(), *SSL_CTX_sess_get_remove_cb()*, and *SSL_CTX_sess_get_get_cb()* allow to retrieve the function pointers of the provided callback functions. If a callback function has not been set, the NULL pointer is returned.

NOTES

In order to allow external session caching, synchronization with the internal session cache is realized via callback functions. Inside these callback functions, session can be saved to disk or put into a database using the *d2i_SSL_SESSION* (3) interface.

The *new_session_cb()* is called, whenever a new session has been negotiated and session caching is enabled (see *SSL_CTX_set_session_cache_mode* (3)). The *new_session_cb()* is passed the **ssl** connection and the ssl session **sess**. If the callback returns **0**, the session will be immediately removed again.

The *remove_session_cb()* is called, whenever the SSL engine removes a session from the internal cache. This happens when the session is removed because it is expired or when a connection was not shutdown cleanly. It also happens for all sessions in the internal session cache when *SSL_CTX_free* (3) is called. The *remove_session_cb()* is passed the **ctx** and the ssl session **sess**. It does not provide any feedback.

The *get_session_cb()* is only called on SSL/TLS servers with the session id proposed by the client. The *get_session_cb()* is always called, also when session caching was disabled. The *get_session_cb()* is passed the **ssl** connection, the session id of length **length** at the memory location **data**. With the parameter **copy** the callback can require the SSL engine to increment the reference count of the SSL_SESSION object, Normally the reference count is not incremented and therefore the session must not be explicitly freed with

SSL_SESSION_free (3).

SEE ALSO

ssl (3), *d2i_SSL_SESSION* (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_CTX_flush_sessions* (3), *SSL_SESSION_free* (3), *SSL_CTX_free* (3)

NAME

SSL_CTX_sessions − access internal session cache

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

struct lhash_st *SSL_CTX_sessions(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_sessions() returns a pointer to the lhash databases containing the internal session cache for **ctx**.

NOTES

The sessions in the internal session cache are kept in an *openssl_lhash*(3) type database. It is possible to directly access this database e.g. for searching. In parallel, the sessions form a linked list which is maintained separately from the *openssl_lhash*(3) operations, so that the database must not be modified directly but by using the *SSL_CTX_add_session*(3) family of functions.

SEE ALSO

ssl(3), *openssl_lhash*(3), *SSL_CTX_add_session*(3), *SSL_CTX_set_session_cache_mode*(3)

NAME

SSL_CTX_set_cert_store, SSL_CTX_get_cert_store − manipulate X509 certificate verification storage

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_cert_store(SSL_CTX *ctx, X509_STORE *store);
X509_STORE *SSL_CTX_get_cert_store(const SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_set_cert_store() sets/replaces the certificate verification storage of **ctx** to/with **store**. If another X509_STORE object is currently set in **ctx**, it will be *X509_STORE_free()*ed.

SSL_CTX_get_cert_store() returns a pointer to the current certificate verification storage.

NOTES

In order to verify the certificates presented by the peer, trusted CA certificates must be accessed. These CA certificates are made available via lookup methods, handled inside the X509_STORE. From the X509_STORE the X509_STORE_CTX used when verifying certificates is created.

Typically the trusted certificate store is handled indirectly via using *SSL_CTX_load_verify_locations* (3). Using the *SSL_CTX_set_cert_store()* and *SSL_CTX_get_cert_store()* functions it is possible to manipulate the X509_STORE object beyond the *SSL_CTX_load_verify_locations* (3) call.

Currently no detailed documentation on how to use the X509_STORE object is available. Not all members of the X509_STORE are used when the verification takes place. So will e.g. the *verify_callback()* be overridden with the *verify_callback()* set via the *SSL_CTX_set_verify* (3) family of functions. This document must therefore be updated when documentation about the X509_STORE object and its handling becomes available.

RETURN VALUES

SSL_CTX_set_cert_store() does not return diagnostic output.

SSL_CTX_get_cert_store() returns the current setting.

SEE ALSO

ssl (3), *SSL_CTX_load_verify_locations* (3), *SSL_CTX_set_verify* (3)

NAME

SSL_CTX_set_cert_verify_callback − set peer certificate verification procedure

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_cert_verify_callback(SSL_CTX *ctx, int (*callback)(X509_STORE_CTX
```

DESCRIPTION

SSL_CTX_set_cert_verify_callback() sets the verification callback function for *ctx*. SSL objects that are created from *ctx* inherit the setting valid at the time when *SSL_new* (3) is called.

NOTES

Whenever a certificate is verified during a SSL/TLS handshake, a verification function is called. If the application does not explicitly specify a verification callback function, the built-in verification function is used. If a verification callback *callback* is specified via *SSL_CTX_set_cert_verify_callback()*, the supplied callback function is called instead. By setting *callback* to NULL, the default behaviour is restored.

When the verification must be performed, *callback* will be called with the arguments callback(X509_STORE_CTX *x509_store_ctx, void *arg). The argument *arg* is specified by the application when setting *callback*.

callback should return 1 to indicate verification success and 0 to indicate verification failure. If SSL_VERIFY_PEER is set and *callback* returns 0, the handshake will fail. As the verification procedure may allow to continue the connection in case of failure (by always returning 1) the verification result must be set in any case using the **error** member of *x509_store_ctx* so that the calling application will be informed about the detailed result of the verification procedure!

Within *x509_store_ctx*, *callback* has access to the *verify_callback* function set using *SSL_CTX_set_verify* (3).

WARNINGS

Do not mix the verification callback described in this function with the **verify_callback** function called during the verification process. The latter is set using the *SSL_CTX_set_verify* (3) family of functions.

Providing a complete verification procedure including certificate purpose settings etc is a complex task. The built-in procedure is quite powerful and in most cases it should be sufficient to modify its behaviour using the **verify_callback** function.

BUGS

RETURN VALUES

SSL_CTX_set_cert_verify_callback() does not provide diagnostic information.

SEE ALSO

ssl (3), *SSL_CTX_set_verify* (3), *SSL_get_verify_result* (3), *SSL_CTX_load_verify_locations* (3)

HISTORY

Previous to OpenSSL 0.9.7, the *arg* argument to **SSL_CTX_set_cert_verify_callback** was ignored, and *callback* was called simply as
int (*callback)(X509_STORE_CTX *) To compile software written for previous versions of OpenSSL, a dummy argument will have to be added to *callback*.

NAME

SSL_CTX_set_cipher_list, SSL_set_cipher_list − choose list of available SSL_CIPHERs

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_CTX_set_cipher_list(SSL_CTX *ctx, const char *str);
int SSL_set_cipher_list(SSL *ssl, const char *str);
```

DESCRIPTION

SSL_CTX_set_cipher_list() sets the list of available ciphers for **ctx** using the control string **str**. The format of the string is described in *openssl_ciphers* (1). The list of ciphers is inherited by all **ssl** objects created from **ctx**.

SSL_set_cipher_list() sets the list of ciphers only for **ssl**.

NOTES

The control string **str** should be universally usable and not depend on details of the library configuration (ciphers compiled in). Thus no syntax checking takes place. Items that are not recognized, because the corresponding ciphers are not compiled in or because they are mistyped, are simply ignored. Failure is only flagged if no ciphers could be collected at all.

It should be noted, that inclusion of a cipher to be used into the list is a necessary condition. On the client side, the inclusion into the list is also sufficient. On the server side, additional restrictions apply. All ciphers have additional requirements. ADH ciphers don't need a certificate, but DH-parameters must have been set. All other ciphers need a corresponding certificate and key.

A RSA cipher can only be chosen, when a RSA certificate is available. RSA export ciphers with a keylength of 512 bits for the RSA key require a temporary 512 bit RSA key, as typically the supplied key has a length of 1024 bit (see *SSL_CTX_set_tmp_rsa_callback* (3)). RSA ciphers using EDH need a certificate and key and additional DH-parameters (see *SSL_CTX_set_tmp_dh_callback* (3)).

A DSA cipher can only be chosen, when a DSA certificate is available. DSA ciphers always use DH key exchange and therefore need DH-parameters (see *SSL_CTX_set_tmp_dh_callback* (3)).

When these conditions are not met for any cipher in the list (e.g. a client only supports export RSA ciphers with a asymmetric key length of 512 bits and the server is not configured to use temporary RSA keys), the ''no shared cipher'' (SSL_R_NO_SHARED_CIPHER) error is generated and the handshake will fail.

RETURN VALUES

SSL_CTX_set_cipher_list() and *SSL_set_cipher_list()* return 1 if any cipher could be selected and 0 on complete failure.

SEE ALSO

ssl (3), *SSL_get_ciphers* (3), *SSL_CTX_use_certificate* (3), *SSL_CTX_set_tmp_rsa_callback* (3), *SSL_CTX_set_tmp_dh_callback* (3), *openssl_ciphers* (1)

NAME

SSL_CTX_set_client_CA_list, SSL_set_client_CA_list, SSL_CTX_add_client_CA, SSL_add_client_CA − set list of CAs sent to the client when requesting a client certificate

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_client_CA_list(SSL_CTX *ctx, STACK_OF(X509_NAME) *list);
void SSL_set_client_CA_list(SSL *s, STACK_OF(X509_NAME) *list);
int SSL_CTX_add_client_CA(SSL_CTX *ctx, X509 *cacert);
int SSL_add_client_CA(SSL *ssl, X509 *cacert);
```

DESCRIPTION

SSL_CTX_set_client_CA_list() sets the **list** of CAs sent to the client when requesting a client certificate for **ctx**.

SSL_set_client_CA_list() sets the **list** of CAs sent to the client when requesting a client certificate for the chosen **ssl**, overriding the setting valid for **ssl**'s SSL_CTX object.

SSL_CTX_add_client_CA() adds the CA name extracted from **cacert** to the list of CAs sent to the client when requesting a client certificate for **ctx**.

SSL_add_client_CA() adds the CA name extracted from **cacert** to the list of CAs sent to the client when requesting a client certificate for the chosen **ssl**, overriding the setting valid for **ssl**'s SSL_CTX object.

NOTES

When a TLS/SSL server requests a client certificate (see ***SSL_CTX_set_verify_options()***), it sends a list of CAs, for which it will accept certificates, to the client.

This list must explicitly be set using *SSL_CTX_set_client_CA_list()* for **ctx** and *SSL_set_client_CA_list()* for the specific **ssl**. The list specified overrides the previous setting. The CAs listed do not become trusted (**list** only contains the names, not the complete certificates); use *SSL_CTX_load_verify_locations* (3) to additionally load them for verification.

If the list of acceptable CAs is compiled in a file, the *SSL_load_client_CA_file* (3) function can be used to help importing the necessary data.

SSL_CTX_add_client_CA() and *SSL_add_client_CA()* can be used to add additional items the list of client CAs. If no list was specified before using *SSL_CTX_set_client_CA_list()* or *SSL_set_client_CA_list()*, a new client CA list for **ctx** or **ssl** (as appropriate) is opened.

These functions are only useful for TLS/SSL servers.

RETURN VALUES

SSL_CTX_set_client_CA_list() and *SSL_set_client_CA_list()* do not return diagnostic information.

SSL_CTX_add_client_CA() and *SSL_add_client_CA()* have the following return values:

1 The operation succeeded.

0 A failure while manipulating the STACK_OF(X509_NAME) object occurred or the X509_NAME could not be extracted from **cacert**. Check the error stack to find out the reason.

EXAMPLES

Scan all certificates in **CAfile** and list them as acceptable CAs:

```
SSL_CTX_set_client_CA_list(ctx,SSL_load_client_CA_file(CAfile));
```

SEE ALSO

ssl (3), *SSL_get_client_CA_list* (3), *SSL_load_client_CA_file* (3), *SSL_CTX_load_verify_locations* (3)

NAME

SSL_CTX_set_client_cert_cb, SSL_CTX_get_client_cert_cb − handle client certificate callback function

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_client_cert_cb(SSL_CTX *ctx, int (*client_cert_cb)(SSL *ssl, X509
int (*SSL_CTX_get_client_cert_cb(SSL_CTX *ctx))(SSL *ssl, X509 **x509, EVP_PKEY **p
int (*client_cert_cb)(SSL *ssl, X509 **x509, EVP_PKEY **pkey);
```

DESCRIPTION

SSL_CTX_set_client_cert_cb() sets the ***client_cert_cb()*** callback, that is called when a client certificate is requested by a server and no certificate was yet set for the SSL object.

When ***client_cert_cb()*** is NULL, no callback function is used.

SSL_CTX_get_client_cert_cb() returns a pointer to the currently set callback function.

client_cert_cb() is the application defined callback. If it wants to set a certificate, a certificate/private key combination must be set using the **x509** and **pkey** arguments and ''1'' must be returned. The certificate will be installed into **ssl**, see the NOTES and BUGS sections. If no certificate should be set, ''0'' has to be returned and no certificate will be sent. A negative return value will suspend the handshake and the handshake function will return immediatly. *SSL_get_error* (3) will return SSL_ERROR_WANT_X509_LOOKUP to indicate, that the handshake was suspended. The next call to the handshake function will again lead to the call of *client_cert_cb()*. It is the job of the *client_cert_cb()* to store information about the state of the last call, if required to continue.

NOTES

During a handshake (or renegotiation) a server may request a certificate from the client. A client certificate must only be sent, when the server did send the request.

When a certificate was set using the *SSL_CTX_use_certificate* (3) family of functions, it will be sent to the server. The TLS standard requires that only a certificate is sent, if it matches the list of acceptable CAs sent by the server. This constraint is violated by the default behavior of the OpenSSL library. Using the callback function it is possible to implement a proper selection routine or to allow a user interaction to choose the certificate to be sent.

If a callback function is defined and no certificate was yet defined for the SSL object, the callback function will be called. If the callback function returns a certificate, the OpenSSL library will try to load the private key and certificate data into the SSL object using the *SSL_use_certificate()* and *SSL_use_private_key()* functions. Thus it will permanently install the certificate and key for this SSL object. It will not be reset by calling *SSL_clear* (3). If the callback returns no certificate, the OpenSSL library will not send a certificate.

BUGS

The *client_cert_cb()* cannot return a complete certificate chain, it can only return one client certificate. If the chain only has a length of 2, the root CA certificate may be omitted according to the TLS standard and thus a standard conforming answer can be sent to the server. For a longer chain, the client must send the complete chain (with the option to leave out the root CA certificate). This can only be accomplished by either adding the intermediate CA certificates into the trusted certificate store for the SSL_CTX object (resulting in having to add CA certificates that otherwise maybe would not be trusted), or by adding the chain certificates using the *SSL_CTX_add_extra_chain_cert* (3) function, which is only available for the SSL_CTX object as a whole and that therefore probably can only apply for one client certificate, making the concept of the callback function (to allow the choice from several certificates) questionable.

Once the SSL object has been used in conjunction with the callback function, the certificate will be set for the SSL object and will not be cleared even when *SSL_clear* (3) is being called. It is therefore mandatory to destroy the SSL object using *SSL_free* (3) and create a new one to return to the previous state.

SEE ALSO

ssl (3), *SSL_CTX_use_certificate* (3), *SSL_CTX_add_extra_chain_cert* (3), *SSL_get_client_CA_list* (3), *SSL_clear* (3), *SSL_free* (3)

NAME

SSL_CTX_set_default_passwd_cb, SSL_CTX_set_default_passwd_cb_userdata − set passwd callback for encrypted PEM file handling

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_default_passwd_cb(SSL_CTX *ctx, pem_password_cb *cb);
void SSL_CTX_set_default_passwd_cb_userdata(SSL_CTX *ctx, void *u);

int pem_passwd_cb(char *buf, int size, int rwflag, void *userdata);
```

DESCRIPTION

SSL_CTX_set_default_passwd_cb() sets the default password callback called when loading/storing a PEM certificate with encryption.

SSL_CTX_set_default_passwd_cb_userdata() sets a pointer to **userdata** which will be provided to the password callback on invocation.

The *pem_passwd_cb()*, which must be provided by the application, hands back the password to be used during decryption. On invocation a pointer to **userdata** is provided. The pem_passwd_cb must write the password into the provided buffer **buf** which is of size **size**. The actual length of the password must be returned to the calling function. **rwflag** indicates whether the callback is used for reading/decryption (rwflag=0) or writing/encryption (rwflag=1).

NOTES

When loading or storing private keys, a password might be supplied to protect the private key. The way this password can be supplied may depend on the application. If only one private key is handled, it can be practical to have *pem_passwd_cb()* handle the password dialog interactively. If several keys have to be handled, it can be practical to ask for the password once, then keep it in memory and use it several times. In the last case, the password could be stored into the **userdata** storage and the *pem_passwd_cb()* only returns the password already stored.

When asking for the password interactively, *pem_passwd_cb()* can use **rwflag** to check, whether an item shall be encrypted (rwflag=1). In this case the password dialog may ask for the same password twice for comparison in order to catch typos, that would make decryption impossible.

Other items in PEM formatting (certificates) can also be encrypted, it is however not usual, as certificate information is considered public.

RETURN VALUES

SSL_CTX_set_default_passwd_cb() and *SSL_CTX_set_default_passwd_cb_userdata()* do not provide diagnostic information.

EXAMPLES

The following example returns the password provided as **userdata** to the calling function. The password is considered to be a '\0' terminated string. If the password does not fit into the buffer, the password is truncated.

```
int pem_passwd_cb(char *buf, int size, int rwflag, void *password)
{
 strncpy(buf, (char *)(password), size);
 buf[size - 1] = '\0';
 return(strlen(buf));
}
```

SEE ALSO

ssl (3), *SSL_CTX_use_certificate* (3)

NAME

SSL_CTX_set_generate_session_id, SSL_set_generate_session_id, SSL_has_matching_session_id − manipulate generation of SSL session IDs (server only)

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

typedef int (*GEN_SESSION_CB)(const SSL *ssl, unsigned char *id,
                              unsigned int *id_len);

int SSL_CTX_set_generate_session_id(SSL_CTX *ctx, GEN_SESSION_CB cb);
int SSL_set_generate_session_id(SSL *ssl, GEN_SESSION_CB, cb);
int SSL_has_matching_session_id(const SSL *ssl, const unsigned char *id,
                                unsigned int id_len);
```

DESCRIPTION

SSL_CTX_set_generate_session_id() sets the callback function for generating new session ids for SSL/TLS sessions for **ctx** to be **cb**.

SSL_set_generate_session_id() sets the callback function for generating new session ids for SSL/TLS sessions for **ssl** to be **cb**.

SSL_has_matching_session_id() checks, whether a session with id **id** (of length **id_len**) is already contained in the internal session cache of the parent context of **ssl**.

NOTES

When a new session is established between client and server, the server generates a session id. The session id is an arbitrary sequence of bytes. The length of the session id is 16 bytes for SSLv2 sessions and between 1 and 32 bytes for SSLv3/TLSv1. The session id is not security critical but must be unique for the server. Additionally, the session id is transmitted in the clear when reusing the session so it must not contain sensitive information.

Without a callback being set, an OpenSSL server will generate a unique session id from pseudo random numbers of the maximum possible length. Using the callback function, the session id can be changed to contain additional information like e.g. a host id in order to improve load balancing or external caching techniques.

The callback function receives a pointer to the memory location to put **id** into and a pointer to the maximum allowed length **id_len**. The buffer at location **id** is only guaranteed to have the size **id_len**. The callback is only allowed to generate a shorter id and reduce **id_len**; the callback **must never** increase **id_len** or write to the location **id** exceeding the given limit.

If a SSLv2 session id is generated and **id_len** is reduced, it will be restored after the callback has finished and the session id will be padded with 0x00. It is not recommended to change the **id_len** for SSLv2 sessions. The callback can use the *SSL_get_version*(3) function to check, whether the session is of type SSLv2.

The location **id** is filled with 0x00 before the callback is called, so the callback may only fill part of the possible length and leave **id_len** untouched while maintaining reproducibility.

Since the sessions must be distinguished, session ids must be unique. Without the callback a random number is used, so that the probability of generating the same session id is extremely small (2^128 possible ids for an SSLv2 session, 2^256 for SSLv3/TLSv1). In order to assure the uniqueness of the generated session id, the callback must call *SSL_has_matching_session_id()* and generate another id if a conflict occurs. If an id conflict is not resolved, the handshake will fail. If the application codes e.g. a unique host id, a unique process number, and a unique sequence number into the session id, uniqueness could easily be achieved without randomness added (it should however be taken care that no confidential information is leaked this way). If the application can not guarantee uniqueness, it is recommended to use the maximum **id_len** and fill in the bytes not used to code special information with random data to avoid collisions.

SSL_has_matching_session_id() will only query the internal session cache, not the external one. Since the session id is generated before the handshake is completed, it is not immediately added to the cache. If another thread is using the same internal session cache, a race condition can occur in that another thread generates the same session id. Collisions can also occur when using an external session cache, since the external cache is not tested with *SSL_has_matching_session_id()* and the same race condition applies.

When calling *SSL_has_matching_session_id()* for an SSLv2 session with reduced **id_len**, the match operation will be performed using the fixed length required and with a 0x00 padded id.

The callback must return 0 if it cannot generate a session id for whatever reason and return 1 on success.

EXAMPLES

The callback function listed will generate a session id with the server id given, and will fill the rest with pseudo random bytes:

```
const char session_id_prefix = "www-18";

#define MAX_SESSION_ID_ATTEMPTS 10
static int generate_session_id(const SSL *ssl, unsigned char *id,
                               unsigned int *id_len)
    {
    unsigned int count = 0;
    const char *version;

    version = SSL_get_version(ssl);
    if (!strcmp(version, "SSLv2"))
        /* we must not change id_len */;

    do      {
            RAND_pseudo_bytes(id, *id_len);
            /* Prefix the session_id with the required prefix. NB: If our
             * prefix is too long, clip it - but there will be worse effects
             * anyway, eg. the server could only possibly create 1 session
             * ID (ie. the prefix!) so all future session negotiations will
             * fail due to conflicts. */
            memcpy(id, session_id_prefix,
                   (strlen(session_id_prefix) < *id_len) ?
                   strlen(session_id_prefix) : *id_len);
            }
    while(SSL_has_matching_session_id(ssl, id, *id_len) &&
            (++count < MAX_SESSION_ID_ATTEMPTS));
    if(count >= MAX_SESSION_ID_ATTEMPTS)
            return 0;
    return 1;
    }
```

RETURN VALUES

SSL_CTX_set_generate_session_id() and *SSL_set_generate_session_id()* always return 1.

SSL_has_matching_session_id() returns 1 if another session with the same id is already in the cache.

SEE ALSO

ssl (3), *SSL_get_version* (3)

HISTORY

SSL_CTX_set_generate_session_id(), *SSL_set_generate_session_id()* and *SSL_has_matching_session_id()* have been introduced in OpenSSL 0.9.7.

NAME

SSL_CTX_set_info_callback, SSL_CTX_get_info_callback, SSL_set_info_callback, SSL_get_info_callback − handle information callback for SSL connections

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_info_callback(SSL_CTX *ctx, void (*callback)());
void (*SSL_CTX_get_info_callback(const SSL_CTX *ctx))();

void SSL_set_info_callback(SSL *ssl, void (*callback)());
void (*SSL_get_info_callback(const SSL *ssl))();
```

DESCRIPTION

SSL_CTX_set_info_callback() sets the **callback** function, that can be used to obtain state information for SSL objects created from **ctx** during connection setup and use. The setting for **ctx** is overridden from the setting for a specific SSL object, if specified. When **callback** is NULL, not callback function is used.

SSL_set_info_callback() sets the **callback** function, that can be used to obtain state information for **ssl** during connection setup and use. When **callback** is NULL, the callback setting currently valid for **ctx** is used.

SSL_CTX_get_info_callback() returns a pointer to the currently set information callback function for **ctx**.

SSL_get_info_callback() returns a pointer to the currently set information callback function for **ssl**.

NOTES

When setting up a connection and during use, it is possible to obtain state information from the SSL/TLS engine. When set, an information callback function is called whenever the state changes, an alert appears, or an error occurs.

The callback function is called as **callback(SSL *ssl, int where, int ret)**. The **where** argument specifies information about where (in which context) the callback function was called. If **ret** is 0, an error condition occurred. If an alert is handled, SSL_CB_ALERT is set and **ret** specifies the alert information.

where is a bitmask made up of the following bits:

SSL_CB_LOOP
: Callback has been called to indicate state change inside a loop.

SSL_CB_EXIT
: Callback has been called to indicate error exit of a handshake function. (May be soft error with retry option for non-blocking setups.)

SSL_CB_READ
: Callback has been called during read operation.

SSL_CB_WRITE
: Callback has been called during write operation.

SSL_CB_ALERT
: Callback has been called due to an alert being sent or received.

SSL_CB_READ_ALERT (SSL_CB_ALERT|SSL_CB_READ)
SSL_CB_WRITE_ALERT (SSL_CB_ALERT|SSL_CB_WRITE)
SSL_CB_ACCEPT_LOOP (SSL_ST_ACCEPT|SSL_CB_LOOP)
SSL_CB_ACCEPT_EXIT (SSL_ST_ACCEPT|SSL_CB_EXIT)
SSL_CB_CONNECT_LOOP (SSL_ST_CONNECT|SSL_CB_LOOP)
SSL_CB_CONNECT_EXIT (SSL_ST_CONNECT|SSL_CB_EXIT)

SSL_CB_HANDSHAKE_START
: Callback has been called because a new handshake is started.

SSL_CB_HANDSHAKE_DONE 0x20
Callback has been called because a handshake is finished.

The current state information can be obtained using the *SSL_state_string* (3) family of functions.

The **ret** information can be evaluated using the *SSL_alert_type_string* (3) family of functions.

RETURN VALUES

SSL_set_info_callback() does not provide diagnostic information.

SSL_get_info_callback() returns the current setting.

EXAMPLES

The following example callback function prints state strings, information about alerts being handled and error messages to the **bio_err** BIO.

```
void apps_ssl_info_callback(SSL *s, int where, int ret)
        {
        const char *str;
        int w;

        w=where& ~SSL_ST_MASK;

        if (w & SSL_ST_CONNECT) str="SSL_connect";
        else if (w & SSL_ST_ACCEPT) str="SSL_accept";
        else str="undefined";

        if (where & SSL_CB_LOOP)
                {
                BIO_printf(bio_err,"%s:%s\n",str,SSL_state_string_long(s));
                }
        else if (where & SSL_CB_ALERT)
                {
                str=(where & SSL_CB_READ)?"read":"write";
                BIO_printf(bio_err,"SSL3 alert %s:%s:%s\n",
                        str,
                        SSL_alert_type_string_long(ret),
                        SSL_alert_desc_string_long(ret));
                }
        else if (where & SSL_CB_EXIT)
                {
                if (ret == 0)
                        BIO_printf(bio_err,"%s:failed in %s\n",
                                str,SSL_state_string_long(s));
                else if (ret < 0)
                        {
                        BIO_printf(bio_err,"%s:error in %s\n",
                                str,SSL_state_string_long(s));
                        }
                }
        }
```

SEE ALSO

ssl (3), *SSL_state_string* (3), *SSL_alert_type_string* (3)

NAME

SSL_CTX_set_max_cert_list, SSL_CTX_get_max_cert_list, SSL_set_max_cert_list, SSL_get_max_cert_list, − manipulate allowed for the peer's certificate chain

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_CTX_set_max_cert_list(SSL_CTX *ctx, long size);
long SSL_CTX_get_max_cert_list(SSL_CTX *ctx);

long SSL_set_max_cert_list(SSL *ssl, long size);
long SSL_get_max_cert_list(SSL *ctx);
```

DESCRIPTION

SSL_CTX_set_max_cert_list() sets the maximum size allowed for the peer's certificate chain for all SSL objects created from **ctx** to be <size> bytes. The SSL objects inherit the setting valid for **ctx** at the time *SSL_new* (3) is being called.

SSL_CTX_get_max_cert_list() returns the currently set maximum size for **ctx**.

SSL_set_max_cert_list() sets the maximum size allowed for the peer's certificate chain for **ssl** to be <size> bytes. This setting stays valid until a new value is set.

SSL_get_max_cert_list() returns the currently set maximum size for **ssl**.

NOTES

During the handshake process, the peer may send a certificate chain. The TLS/SSL standard does not give any maximum size of the certificate chain. The OpenSSL library handles incoming data by a dynamically allocated buffer. In order to prevent this buffer from growing without bounds due to data received from a faulty or malicious peer, a maximum size for the certificate chain is set.

The default value for the maximum certificate chain size is 100kB (30kB on the 16bit DOS platform). This should be sufficient for usual certificate chains (OpenSSL's default maximum chain length is 10, see *SSL_CTX_set_verify* (3), and certificates without special extensions have a typical size of 1−2kB).

For special applications it can be necessary to extend the maximum certificate chain size allowed to be sent by the peer, see e.g. the work on ''Internet X.509 Public Key Infrastructure Proxy Certificate Profile'' and ''TLS Delegation Protocol'' at http://www.ietf.org/ and http://www.globus.org/ .

Under normal conditions it should never be necessary to set a value smaller than the default, as the buffer is handled dynamically and only uses the memory actually required by the data sent by the peer.

If the maximum certificate chain size allowed is exceeded, the handshake will fail with a SSL_R_EXCESSIVE_MESSAGE_SIZE error.

RETURN VALUES

SSL_CTX_set_max_cert_list() and *SSL_set_max_cert_list()* return the previously set value.

SSL_CTX_get_max_cert_list() and *SSL_get_max_cert_list()* return the currently set value.

SEE ALSO

ssl (3), *SSL_new* (3), *SSL_CTX_set_verify* (3)

HISTORY

SSL*_set/*get_max_cert_list()* have been introduced in OpenSSL 0.9.7.

NAME

SSL_CTX_set_mode, SSL_set_mode, SSL_CTX_get_mode, SSL_get_mode − manipulate SSL engine mode

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_CTX_set_mode(SSL_CTX *ctx, long mode);
long SSL_set_mode(SSL *ssl, long mode);

long SSL_CTX_get_mode(SSL_CTX *ctx);
long SSL_get_mode(SSL *ssl);
```

DESCRIPTION

SSL_CTX_set_mode() adds the mode set via bitmask in **mode** to **ctx**. Options already set before are not cleared.

SSL_set_mode() adds the mode set via bitmask in **mode** to **ssl**. Options already set before are not cleared.

SSL_CTX_get_mode() returns the mode set for **ctx**.

SSL_get_mode() returns the mode set for **ssl**.

NOTES

The following mode changes are available:

SSL_MODE_ENABLE_PARTIAL_WRITE

Allow SSL_write(..., n) to return r with 0 < r < n (i.e. report success when just a single record has been written). When not set (the default), *SSL_write()* will only report success once the complete chunk was written. Once *SSL_write()* returns with r, r bytes have been successfully written and the next call to *SSL_write()* must only send the n−r bytes left, imitating the behaviour of *write()*.

SSL_MODE_ACCEPT_MOVING_WRITE_BUFFER

Make it possible to retry *SSL_write()* with changed buffer location (the buffer contents must stay the same). This is not the default to avoid the misconception that non-blocking *SSL_write()* behaves like non-blocking *write()*.

SSL_MODE_AUTO_RETRY

Never bother the application with retries if the transport is blocking. If a renegotiation take place during normal operation, a *SSL_read* (3) or *SSL_write* (3) would return with −1 and indicate the need to retry with SSL_ERROR_WANT_READ. In a non-blocking environment applications must be prepared to handle incomplete read/write operations. In a blocking environment, applications are not always prepared to deal with read/write operations returning without success report. The flag SSL_MODE_AUTO_RETRY will cause read/write operations to only return after the handshake and successful completion.

RETURN VALUES

SSL_CTX_set_mode() and *SSL_set_mode()* return the new mode bitmask after adding **mode**.

SSL_CTX_get_mode() and *SSL_get_mode()* return the current bitmask.

SEE ALSO

ssl (3), *SSL_read* (3), *SSL_write* (3)

HISTORY

SSL_MODE_AUTO_RETRY as been added in OpenSSL 0.9.6.

NAME

SSL_CTX_set_msg_callback, SSL_CTX_set_msg_callback_arg, SSL_set_msg_callback, SSL_get_msg_callback_arg − install callback for observing protocol messages

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_msg_callback(SSL_CTX *ctx, void (*cb)(int write_p, int version, in
void SSL_CTX_set_msg_callback_arg(SSL_CTX *ctx, void *arg);

void SSL_set_msg_callback(SSL_CTX *ctx, void (*cb)(int write_p, int version, int co
void SSL_set_msg_callback_arg(SSL_CTX *ctx, void *arg);
```

DESCRIPTION

SSL_CTX_set_msg_callback() or *SSL_set_msg_callback()* can be used to define a message callback function *cb* for observing all SSL/TLS protocol messages (such as handshake messages) that are received or sent. *SSL_CTX_set_msg_callback_arg()* and *SSL_set_msg_callback_arg()* can be used to set argument *arg* to the callback function, which is available for arbitrary application use.

SSL_CTX_set_msg_callback() and *SSL_CTX_set_msg_callback_arg()* specify default settings that will be copied to new **SSL** objects by *SSL_new*(3). *SSL_set_msg_callback()* and *SSL_set_msg_callback_arg()* modify the actual settings of an **SSL** object. Using a **0** pointer for *cb* disables the message callback.

When *cb* is called by the SSL/TLS library for a protocol message, the function arguments have the following meaning:

write_p
: This flag is **0** when a protocol message has been received and **1** when a protocol message has been sent.

version
: The protocol version according to which the protocol message is interpreted by the library. Currently, this is one of **SSL2_VERSION**, **SSL3_VERSION** and **TLS1_VERSION** (for SSL 2.0, SSL 3.0 and TLS 1.0, respectively).

content_type
: In the case of SSL 2.0, this is always **0**. In the case of SSL 3.0 or TLS 1.0, this is one of the **Content-Type** values defined in the protocol specification (**change_cipher_spec(20)**, **alert(21)**, **handshake(22)**; but never **application_data(23)** because the callback will only be called for protocol messages).

buf, *len*
: *buf* points to a buffer containing the protocol message, which consists of *len* bytes. The buffer is no longer valid after the callback function has returned.

ssl The **SSL** object that received or sent the message.

arg The user-defined argument optionally defined by *SSL_CTX_set_msg_callback_arg()* or *SSL_set_msg_callback_arg()*.

NOTES

Protocol messages are passed to the callback function after decryption and fragment collection where applicable. (Thus record boundaries are not visible.)

If processing a received protocol message results in an error, the callback function may not be called. For example, the callback function will never see messages that are considered too large to be processed.

Due to automatic protocol version negotiation, *version* is not necessarily the protocol version used by the sender of the message: If a TLS 1.0 ClientHello message is received by an SSL 3.0−only server, *version* will be **SSL3_VERSION**.

SEE ALSO

ssl (3), *SSL_new* (3)

HISTORY

SSL_CTX_set_msg_callback(), *SSL_CTX_set_msg_callback_arg()*, *SSL_set_msg_callback()* and *SSL_get_msg_callback_arg()* were added in OpenSSL 0.9.7.

NAME

SSL_CTX_set_options, SSL_set_options, SSL_CTX_get_options, SSL_get_options − manipulate SSL engine options

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_CTX_set_options(SSL_CTX *ctx, long options);
long SSL_set_options(SSL *ssl, long options);

long SSL_CTX_get_options(SSL_CTX *ctx);
long SSL_get_options(SSL *ssl);
```

DESCRIPTION

SSL_CTX_set_options() adds the options set via bitmask in **options** to **ctx**. Options already set before are not cleared!

SSL_set_options() adds the options set via bitmask in **options** to **ssl**. Options already set before are not cleared!

SSL_CTX_get_options() returns the options set for **ctx**.

SSL_get_options() returns the options set for **ssl**.

NOTES

The behaviour of the SSL library can be changed by setting several options. The options are coded as bitmasks and can be combined by a logical **or** operation (|). Options can only be added but can never be reset.

SSL_CTX_set_options() and *SSL_set_options()* affect the (external) protocol behaviour of the SSL library. The (internal) behaviour of the API can be changed by using the similar *SSL_CTX_set_mode* (3) and *SSL_set_mode()* functions.

During a handshake, the option settings of the SSL object are used. When a new SSL object is created from a context using *SSL_new()*, the current option setting is copied. Changes to **ctx** do not affect already created SSL objects. *SSL_clear()* does not affect the settings.

The following **bug workaround** options are available:

SSL_OP_MICROSOFT_SESS_ID_BUG
: www.microsoft.com − when talking SSLv2, if session-id reuse is performed, the session-id passed back in the server-finished message is different from the one decided upon.

SSL_OP_NETSCAPE_CHALLENGE_BUG
: Netscape−Commerce/1.12, when talking SSLv2, accepts a 32 byte challenge but then appears to only use 16 bytes when generating the encryption keys. Using 16 bytes is ok but it should be ok to use 32. According to the SSLv3 spec, one should use 32 bytes for the challenge when operating in SSLv2/v3 compatibility mode, but as mentioned above, this breaks this server so 16 bytes is the way to go.

SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG
: ssl3.netscape.com:443, first a connection is established with RC4−MD5. If it is then resumed, we end up using DES−CBC3−SHA. It should be RC4−MD5 according to 7.6.1.3, 'cipher_suite'.

  Netscape−Enterprise/2.01 (https://merchant.netscape.com) has this bug. It only really shows up when connecting via SSLv2/v3 then reconnecting via SSLv3. The cipher list changes....

  NEW INFORMATION. Try connecting with a cipher list of just DES−CBC−SHA:RC4−MD5. For some weird reason, each new connection uses RC4−MD5, but a re-connect tries to use DES−CBC−SHA. So netscape, when doing a re−connect, always takes the first cipher in the cipher list.

SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG
: ...

SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER

...

SSL_OP_MSIE_SSLV2_RSA_PADDING

As of OpenSSL 0.9.7h and 0.9.8a, this option has no effect.

SSL_OP_SSLEAY_080_CLIENT_DH_BUG

...

SSL_OP_TLS_D5_BUG

...

SSL_OP_TLS_BLOCK_PADDING_BUG

...

SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS

Disables a countermeasure against a SSL 3.0/TLS 1.0 protocol vulnerability affecting CBC ciphers, which cannot be handled by some broken SSL implementations. This option has no effect for connections using other ciphers.

SSL_OP_ALL

All of the above bug workarounds.

It is usually safe to use **SSL_OP_ALL** to enable the bug workaround options if compatibility with somewhat broken implementations is desired.

The following **modifying** options are available:

SSL_OP_TLS_ROLLBACK_BUG

Disable version rollback attack detection.

During the client key exchange, the client must send the same information about acceptable SSL/TLS protocol levels as during the first hello. Some clients violate this rule by adapting to the server's answer. (Example: the client sends a SSLv2 hello and accepts up to SSLv3.1=TLSv1, the server only understands up to SSLv3. In this case the client must still use the same SSLv3.1=TLSv1 announcement. Some clients step down to SSLv3 with respect to the server's answer and violate the version rollback protection.)

SSL_OP_SINGLE_DH_USE

Always create a new key when using temporary/ephemeral DH parameters (see *SSL_CTX_set_tmp_dh_callback*(3)). This option must be used to prevent small subgroup attacks, when the DH parameters were not generated using ''strong'' primes (e.g. when using DSA−parameters, see *openssl_dhparam*(1)). If ''strong'' primes were used, it is not strictly necessary to generate a new DH key during each handshake but it is also recommended. **SSL_OP_SINGLE_DH_USE** should therefore be enabled whenever temporary/ephemeral DH parameters are used.

SSL_OP_EPHEMERAL_RSA

Always use ephemeral (temporary) RSA key when doing RSA operations (see *SSL_CTX_set_tmp_rsa_callback*(3)). According to the specifications this is only done, when a RSA key can only be used for signature operations (namely under export ciphers with restricted RSA keylength). By setting this option, ephemeral RSA keys are always used. This option breaks compatibility with the SSL/TLS specifications and may lead to interoperability problems with clients and should therefore never be used. Ciphers with EDH (ephemeral Diffie−Hellman) key exchange should be used instead.

SSL_OP_CIPHER_SERVER_PREFERENCE

When choosing a cipher, use the server's preferences instead of the client preferences. When not set, the SSL server will always follow the clients preferences. When set, the SSLv3/TLSv1 server will choose following its own preferences. Because of the different protocol, for SSLv2 the server will send its list of preferences to the client and the client chooses.

SSL_OP_PKCS1_CHECK_1

...

SSL_OP_PKCS1_CHECK_2

...

SSL_OP_NETSCAPE_CA_DN_BUG

If we accept a netscape connection, demand a client cert, have a non-self-signed CA which does not have its CA in netscape, and the browser has a cert, it will crash/hang. Works for 3.x and 4.xbeta

SSL_OP_NETSCAPE_DEMO_CIPHER_CHANGE_BUG

...

SSL_OP_NO_SSLv2

Do not use the SSLv2 protocol.

SSL_OP_NO_SSLv3

Do not use the SSLv3 protocol.

SSL_OP_NO_TLSv1

Do not use the TLSv1 protocol.

SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION

When performing renegotiation as a server, always start a new session (i.e., session resumption requests are only accepted in the initial handshake). This option is not needed for clients.

SSL_OP_NO_TICKET

Normally clients and servers will, where possible, transparently make use of RFC4507bis tickets for stateless session resumption.

If this option is set this functionality is disabled and tickets will not be used by clients or servers.

RETURN VALUES

SSL_CTX_set_options() and *SSL_set_options()* return the new options bitmask after adding **options**.

SSL_CTX_get_options() and *SSL_get_options()* return the current bitmask.

SEE ALSO

ssl (3), *SSL_new* (3), *SSL_clear* (3), *SSL_CTX_set_tmp_dh_callback* (3), *SSL_CTX_set_tmp_rsa_callback* (3), *openssl_dhparam* (1)

HISTORY

SSL_OP_CIPHER_SERVER_PREFERENCE and **SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION** have been added in OpenSSL 0.9.7.

SSL_OP_TLS_ROLLBACK_BUG has been added in OpenSSL 0.9.6 and was automatically enabled with **SSL_OP_ALL**. As of 0.9.7, it is no longer included in **SSL_OP_ALL** and must be explicitly set.

SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS has been added in OpenSSL 0.9.6e. Versions up to OpenSSL 0.9.6c do not include the countermeasure that can be disabled with this option (in OpenSSL 0.9.6d, it was always enabled).

NAME

SSL_CTX_set_quiet_shutdown, SSL_CTX_get_quiet_shutdown, SSL_set_quiet_shutdown, SSL_get_quiet_shutdown − manipulate shutdown behaviour

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_quiet_shutdown(SSL_CTX *ctx, int mode);
int SSL_CTX_get_quiet_shutdown(const SSL_CTX *ctx);

void SSL_set_quiet_shutdown(SSL *ssl, int mode);
int SSL_get_quiet_shutdown(const SSL *ssl);
```

DESCRIPTION

SSL_CTX_set_quiet_shutdown() sets the ‘‘quiet shutdown’’ flag for **ctx** to be **mode**. SSL objects created from **ctx** inherit the **mode** valid at the time *SSL_new* (3) is called. **mode** may be 0 or 1.

SSL_CTX_get_quiet_shutdown() returns the ‘‘quiet shutdown’’ setting of **ctx**.

SSL_set_quiet_shutdown() sets the ‘‘quiet shutdown’’ flag for **ssl** to be **mode**. The setting stays valid until **ssl** is removed with *SSL_free* (3) or *SSL_set_quiet_shutdown()* is called again. It is not changed when *SSL_clear* (3) is called. **mode** may be 0 or 1.

SSL_get_quiet_shutdown() returns the ‘‘quiet shutdown’’ setting of **ssl**.

NOTES

Normally when a SSL connection is finished, the parties must send out ‘‘close notify’’ alert messages using *SSL_shutdown* (3) for a clean shutdown.

When setting the ‘‘quiet shutdown’’ flag to 1, *SSL_shutdown* (3) will set the internal flags to SSL_SENT_SHUTDOWN|SSL_RECEIVED_SHUTDOWN. (*SSL_shutdown* (3) then behaves like *SSL_set_shutdown* (3) called with SSL_SENT_SHUTDOWN|SSL_RECEIVED_SHUTDOWN.) The session is thus considered to be shutdown, but no ‘‘close notify’’ alert is sent to the peer. This behaviour violates the TLS standard.

The default is normal shutdown behaviour as described by the TLS standard.

RETURN VALUES

SSL_CTX_set_quiet_shutdown() and *SSL_set_quiet_shutdown()* do not return diagnostic information.

SSL_CTX_get_quiet_shutdown() and SSL_get_quiet_shutdown return the current setting.

SEE ALSO

ssl (3), *SSL_shutdown* (3), *SSL_set_shutdown* (3), *SSL_new* (3), *SSL_clear* (3), *SSL_free* (3)

NAME

SSL_CTX_set_session_cache_mode, SSL_CTX_get_session_cache_mode − enable/disable session caching

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_CTX_set_session_cache_mode(SSL_CTX ctx, long mode);
long SSL_CTX_get_session_cache_mode(SSL_CTX ctx);
```

DESCRIPTION

SSL_CTX_set_session_cache_mode() enables/disables session caching by setting the operational mode for **ctx** to <mode>.

SSL_CTX_get_session_cache_mode() returns the currently used cache mode.

NOTES

The OpenSSL library can store/retrieve SSL/TLS sessions for later reuse. The sessions can be held in memory for each **ctx**, if more than one SSL_CTX object is being maintained, the sessions are unique for each SSL_CTX object.

In order to reuse a session, a client must send the session's id to the server. It can only send exactly one id. The server then either agrees to reuse the session or it starts a full handshake (to create a new session).

A server will lookup up the session in its internal session storage. If the session is not found in internal storage or lookups for the internal storage have been deactivated (SSL_SESS_CACHE_NO_INTERNAL_LOOKUP), the server will try the external storage if available.

Since a client may try to reuse a session intended for use in a different context, the session id context must be set by the server (see *SSL_CTX_set_session_id_context* (3)).

The following session cache modes and modifiers are available:

SSL_SESS_CACHE_OFF
: No session caching for client or server takes place.

SSL_SESS_CACHE_CLIENT
: Client sessions are added to the session cache. As there is no reliable way for the OpenSSL library to know whether a session should be reused or which session to choose (due to the abstract BIO layer the SSL engine does not have details about the connection), the application must select the session to be reused by using the *SSL_set_session* (3) function. This option is not activated by default.

SSL_SESS_CACHE_SERVER
: Server sessions are added to the session cache. When a client proposes a session to be reused, the server looks for the corresponding session in (first) the internal session cache (unless SSL_SESS_CACHE_NO_INTERNAL_LOOKUP is set), then (second) in the external cache if available. If the session is found, the server will try to reuse the session. This is the default.

SSL_SESS_CACHE_BOTH
: Enable both SSL_SESS_CACHE_CLIENT and SSL_SESS_CACHE_SERVER at the same time.

SSL_SESS_CACHE_NO_AUTO_CLEAR
: Normally the session cache is checked for expired sessions every 255 connections using the *SSL_CTX_flush_sessions* (3) function. Since this may lead to a delay which cannot be controlled, the automatic flushing may be disabled and *SSL_CTX_flush_sessions* (3) can be called explicitly by the application.

SSL_SESS_CACHE_NO_INTERNAL_LOOKUP
: By setting this flag, session-resume operations in an SSL/TLS server will not automatically look up sessions in the internal cache, even if sessions are automatically stored there. If external session caching callbacks are in use, this flag guarantees that all lookups are directed to the external cache. As

automatic lookup only applies for SSL/TLS servers, the flag has no effect on clients.

SSL_SESS_CACHE_NO_INTERNAL_STORE
: Depending on the presence of SSL_SESS_CACHE_CLIENT and/or SSL_SESS_CACHE_SERVER, sessions negotiated in an SSL/TLS handshake may be cached for possible reuse. Normally a new session is added to the internal cache as well as any external session caching (callback) that is configured for the SSL_CTX. This flag will prevent sessions being stored in the internal cache (though the application can add them manually using *SSL_CTX_add_session*(3)). Note: in any SSL/TLS servers where external caching is configured, any successful session lookups in the external cache (ie. for session-resume requests) would normally be copied into the local cache before processing continues − this flag prevents these additions to the internal cache as well.

SSL_SESS_CACHE_NO_INTERNAL
: Enable both SSL_SESS_CACHE_NO_INTERNAL_LOOKUP and SSL_SESS_CACHE_NO_INTERNAL_STORE at the same time.

The default mode is SSL_SESS_CACHE_SERVER.

RETURN VALUES

SSL_CTX_set_session_cache_mode() returns the previously set cache mode.

SSL_CTX_get_session_cache_mode() returns the currently set cache mode.

SEE ALSO

ssl(3), *SSL_set_session*(3), *SSL_session_reused*(3), *SSL_CTX_add_session*(3), *SSL_CTX_sess_number*(3), *SSL_CTX_sess_set_cache_size*(3), *SSL_CTX_sess_set_get_cb*(3), *SSL_CTX_set_session_id_context*(3), *SSL_CTX_set_timeout*(3), *SSL_CTX_flush_sessions*(3)

HISTORY

SSL_SESS_CACHE_NO_INTERNAL_STORE and SSL_SESS_CACHE_NO_INTERNAL were introduced in OpenSSL 0.9.6h.

NAME

SSL_CTX_set_session_id_context, SSL_set_session_id_context − set context within which session can be reused (server side only)

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_CTX_set_session_id_context(SSL_CTX *ctx, const unsigned char *sid_ctx,
                                   unsigned int sid_ctx_len);
int SSL_set_session_id_context(SSL *ssl, const unsigned char *sid_ctx,
                               unsigned int sid_ctx_len);
```

DESCRIPTION

SSL_CTX_set_session_id_context() sets the context **sid_ctx** of length **sid_ctx_len** within which a session can be reused for the **ctx** object.

SSL_set_session_id_context() sets the context **sid_ctx** of length **sid_ctx_len** within which a session can be reused for the **ssl** object.

NOTES

Sessions are generated within a certain context. When exporting/importing sessions with **i2d_SSL_SESSION**/**d2i_SSL_SESSION** it would be possible, to re-import a session generated from another context (e.g. another application), which might lead to malfunctions. Therefore each application must set its own session id context **sid_ctx** which is used to distinguish the contexts and is stored in exported sessions. The **sid_ctx** can be any kind of binary data with a given length, it is therefore possible to use e.g. the name of the application and/or the hostname and/or service name ...

The session id context becomes part of the session. The session id context is set by the SSL/TLS server. The *SSL_CTX_set_session_id_context()* and *SSL_set_session_id_context()* functions are therefore only useful on the server side.

OpenSSL clients will check the session id context returned by the server when reusing a session.

The maximum length of the **sid_ctx** is limited to **SSL_MAX_SSL_SESSION_ID_LENGTH**.

WARNINGS

If the session id context is not set on an SSL/TLS server and client certificates are used, stored sessions will not be reused but a fatal error will be flagged and the handshake will fail.

If a server returns a different session id context to an OpenSSL client when reusing a session, an error will be flagged and the handshake will fail. OpenSSL servers will always return the correct session id context, as an OpenSSL server checks the session id context itself before reusing a session as described above.

RETURN VALUES

SSL_CTX_set_session_id_context() and *SSL_set_session_id_context()* return the following values:

0 The length **sid_ctx_len** of the session id context **sid_ctx** exceeded the maximum allowed length of **SSL_MAX_SSL_SESSION_ID_LENGTH**. The error is logged to the error stack.

1 The operation succeeded.

SEE ALSO

ssl (3)

NAME

SSL_CTX_set_ssl_version, SSL_set_ssl_method, SSL_get_ssl_method − choose a new TLS/SSL method

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_CTX_set_ssl_version(SSL_CTX *ctx, const SSL_METHOD *method);
int SSL_set_ssl_method(SSL *s, const SSL_METHOD *method);
const SSL_METHOD *SSL_get_ssl_method(SSL *ssl);
```

DESCRIPTION

SSL_CTX_set_ssl_version() sets a new default TLS/SSL **method** for SSL objects newly created from this **ctx**. SSL objects already created with *SSL_new*(3) are not affected, except when *SSL_clear*(3) is being called.

SSL_set_ssl_method() sets a new TLS/SSL **method** for a particular **ssl** object. It may be reset, when *SSL_clear()* is called.

SSL_get_ssl_method() returns a function pointer to the TLS/SSL method set in **ssl**.

NOTES

The available **method** choices are described in *SSL_CTX_new*(3).

When *SSL_clear*(3) is called and no session is connected to an SSL object, the method of the SSL object is reset to the method currently set in the corresponding SSL_CTX object.

RETURN VALUES

The following return values can occur for *SSL_CTX_set_ssl_version()* and *SSL_set_ssl_method()*:

0 The new choice failed, check the error stack to find out the reason.

1 The operation succeeded.

SEE ALSO

SSL_CTX_new(3), *SSL_new*(3), *SSL_clear*(3), *ssl*(3), *SSL_set_connect_state*(3)

NAME

SSL_CTX_set_timeout, SSL_CTX_get_timeout − manipulate timeout values for session caching

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_CTX_set_timeout(SSL_CTX *ctx, long t);
long SSL_CTX_get_timeout(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_set_timeout() sets the timeout for newly created sessions for **ctx** to **t**. The timeout value **t** must be given in seconds.

SSL_CTX_get_timeout() returns the currently set timeout value for **ctx**.

NOTES

Whenever a new session is created, it is assigned a maximum lifetime. This lifetime is specified by storing the creation time of the session and the timeout value valid at this time. If the actual time is later than creation time plus timeout, the session is not reused.

Due to this realization, all sessions behave according to the timeout value valid at the time of the session negotiation. Changes of the timeout value do not affect already established sessions.

The expiration time of a single session can be modified using the *SSL_SESSION_get_time*(3) family of functions.

Expired sessions are removed from the internal session cache, whenever *SSL_CTX_flush_sessions*(3) is called, either directly by the application or automatically (see *SSL_CTX_set_session_cache_mode*(3))

The default value for session timeout is decided on a per protocol basis, see *SSL_get_default_timeout*(3). All currently supported protocols have the same default timeout value of 300 seconds.

RETURN VALUES

SSL_CTX_set_timeout() returns the previously set timeout value.

SSL_CTX_get_timeout() returns the currently set timeout value.

SEE ALSO

ssl(3), *SSL_CTX_set_session_cache_mode*(3), *SSL_SESSION_get_time*(3), *SSL_CTX_flush_sessions*(3), *SSL_get_default_timeout*(3)

NAME

SSL_CTX_set_tmp_dh_callback, SSL_CTX_set_tmp_dh, SSL_set_tmp_dh_callback, SSL_set_tmp_dh − handle DH keys for ephemeral key exchange

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_tmp_dh_callback(SSL_CTX *ctx,
          DH *(*tmp_dh_callback)(SSL *ssl, int is_export, int keylength));
long SSL_CTX_set_tmp_dh(SSL_CTX *ctx, DH *dh);

void SSL_set_tmp_dh_callback(SSL_CTX *ctx,
          DH *(*tmp_dh_callback)(SSL *ssl, int is_export, int keylength));
long SSL_set_tmp_dh(SSL *ssl, DH *dh)

DH *(*tmp_dh_callback)(SSL *ssl, int is_export, int keylength));
```

DESCRIPTION

SSL_CTX_set_tmp_dh_callback() sets the callback function for **ctx** to be used when a DH parameters are required to **tmp_dh_callback**. The callback is inherited by all **ssl** objects created from **ctx**.

SSL_CTX_set_tmp_dh() sets DH parameters to be used to be **dh**. The key is inherited by all **ssl** objects created from **ctx**.

SSL_set_tmp_dh_callback() sets the callback only for **ssl**.

SSL_set_tmp_dh() sets the parameters only for **ssl**.

These functions apply to SSL/TLS servers only.

NOTES

When using a cipher with RSA authentication, an ephemeral DH key exchange can take place. Ciphers with DSA keys always use ephemeral DH keys as well. In these cases, the session data are negotiated using the ephemeral/temporary DH key and the key supplied and certified by the certificate chain is only used for signing. Anonymous ciphers (without a permanent server key) also use ephemeral DH keys.

Using ephemeral DH key exchange yields forward secrecy, as the connection can only be decrypted, when the DH key is known. By generating a temporary DH key inside the server application that is lost when the application is left, it becomes impossible for an attacker to decrypt past sessions, even if he gets hold of the normal (certified) key, as this key was only used for signing.

In order to perform a DH key exchange the server must use a DH group (DH parameters) and generate a DH key. The server will always generate a new DH key during the negotiation, when the DH parameters are supplied via callback and/or when the SSL_OP_SINGLE_DH_USE option of *SSL_CTX_set_options*(3) is set. It will immediately create a DH key, when DH parameters are supplied via *SSL_CTX_set_tmp_dh()* and SSL_OP_SINGLE_DH_USE is not set. In this case, it may happen that a key is generated on initialization without later being needed, while on the other hand the computer time during the negotiation is being saved.

If ‘‘strong’’ primes were used to generate the DH parameters, it is not strictly necessary to generate a new key for each handshake but it does improve forward secrecy. If it is not assured, that ‘‘strong’’ primes were used (see especially the section about DSA parameters below), SSL_OP_SINGLE_DH_USE must be used in order to prevent small subgroup attacks. Always using SSL_OP_SINGLE_DH_USE has an impact on the computer time needed during negotiation, but it is not very large, so application authors/users should consider to always enable this option.

As generating DH parameters is extremely time consuming, an application should not generate the parameters on the fly but supply the parameters. DH parameters can be reused, as the actual key is newly generated during the negotiation. The risk in reusing DH parameters is that an attacker may specialize on a very often used DH group. Applications should therefore generate their own DH parameters during the

installation process using the openssl *openssl_dhparam*(1) application. In order to reduce the computer time needed for this generation, it is possible to use DSA parameters instead (see *openssl_dhparam*(1)), but in this case SSL_OP_SINGLE_DH_USE is mandatory.

Application authors may compile in DH parameters. Files dh512.pem, dh1024.pem, dh2048.pem, and dh4096 in the 'apps' directory of current version of the OpenSSL distribution contain the 'SKIP' DH parameters, which use safe primes and were generated verifiably pseudo−randomly. These files can be converted into C code using the **−C** option of the *openssl_dhparam*(1) application. Authors may also generate their own set of parameters using *openssl_dhparam*(1), but a user may not be sure how the parameters were generated. The generation of DH parameters during installation is therefore recommended.

An application may either directly specify the DH parameters or can supply the DH parameters via a callback function. The callback approach has the advantage, that the callback may supply DH parameters for different key lengths.

The **tmp_dh_callback** is called with the **keylength** needed and the **is_export** information. The **is_export** flag is set, when the ephemeral DH key exchange is performed with an export cipher.

EXAMPLES

Handle DH parameters for key lengths of 512 and 1024 bits. (Error handling partly left out.)

```
...
/* Set up ephemeral DH stuff */
DH *dh_512 = NULL;
DH *dh_1024 = NULL;
FILE *paramfile;

...
/* "openssl dhparam -out dh_param_512.pem -2 512" */
paramfile = fopen("dh_param_512.pem", "r");
if (paramfile) {
  dh_512 = PEM_read_DHparams(paramfile, NULL, NULL, NULL);
  fclose(paramfile);
}
/* "openssl dhparam -out dh_param_1024.pem -2 1024" */
paramfile = fopen("dh_param_1024.pem", "r");
if (paramfile) {
  dh_1024 = PEM_read_DHparams(paramfile, NULL, NULL, NULL);
  fclose(paramfile);
}
...
/* "openssl dhparam -C -2 512" etc... */
DH *get_dh512() { ... }
DH *get_dh1024() { ... }

DH *tmp_dh_callback(SSL *s, int is_export, int keylength)
{
   DH *dh_tmp=NULL;
```

```
        switch (keylength) {
        case 512:
          if (!dh_512)
            dh_512 = get_dh512();
          dh_tmp = dh_512;
          break;
        case 1024:
          if (!dh_1024)
            dh_1024 = get_dh1024();
          dh_tmp = dh_1024;
          break;
        default:
          /* Generating a key on the fly is very costly, so use what is there */
          setup_dh_parameters_like_above();
        }
        return(dh_tmp);
    }
```

RETURN VALUES

SSL_CTX_set_tmp_dh_callback() and *SSL_set_tmp_dh_callback()* do not return diagnostic output.

SSL_CTX_set_tmp_dh() and *SSL_set_tmp_dh()* do return 1 on success and 0 on failure. Check the error queue to find out the reason of failure.

SEE ALSO

ssl (3), *SSL_CTX_set_cipher_list* (3), *SSL_CTX_set_tmp_rsa_callback* (3), *SSL_CTX_set_options* (3), *openssl_ciphers* (1), *openssl_dhparam* (1)

NAME

SSL_CTX_set_tmp_rsa_callback, SSL_CTX_set_tmp_rsa, SSL_CTX_need_tmp_rsa, SSL_set_tmp_rsa_callback, SSL_set_tmp_rsa, SSL_need_tmp_rsa − handle RSA keys for ephemeral key exchange

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_tmp_rsa_callback(SSL_CTX *ctx,
          RSA *(*tmp_rsa_callback)(SSL *ssl, int is_export, int keylength));
long SSL_CTX_set_tmp_rsa(SSL_CTX *ctx, RSA *rsa);
long SSL_CTX_need_tmp_rsa(SSL_CTX *ctx);

void SSL_set_tmp_rsa_callback(SSL_CTX *ctx,
          RSA *(*tmp_rsa_callback)(SSL *ssl, int is_export, int keylength));
long SSL_set_tmp_rsa(SSL *ssl, RSA *rsa)
long SSL_need_tmp_rsa(SSL *ssl)

RSA *(*tmp_rsa_callback)(SSL *ssl, int is_export, int keylength);
```

DESCRIPTION

SSL_CTX_set_tmp_rsa_callback() sets the callback function for **ctx** to be used when a temporary/ephemeral RSA key is required to **tmp_rsa_callback**. The callback is inherited by all SSL objects newly created from **ctx** with <*SSL_new*(3)|*SSL_new*(3)>. Already created SSL objects are not affected.

SSL_CTX_set_tmp_rsa() sets the temporary/ephemeral RSA key to be used to be **rsa**. The key is inherited by all SSL objects newly created from **ctx** with <*SSL_new*(3)|*SSL_new*(3)>. Already created SSL objects are not affected.

SSL_CTX_need_tmp_rsa() returns 1, if a temporary/ephemeral RSA key is needed for RSA-based strength-limited 'exportable' ciphersuites because a RSA key with a keysize larger than 512 bits is installed.

SSL_set_tmp_rsa_callback() sets the callback only for **ssl**.

SSL_set_tmp_rsa() sets the key only for **ssl**.

SSL_need_tmp_rsa() returns 1, if a temporary/ephemeral RSA key is needed, for RSA-based strength-limited 'exportable' ciphersuites because a RSA key with a keysize larger than 512 bits is installed.

These functions apply to SSL/TLS servers only.

NOTES

When using a cipher with RSA authentication, an ephemeral RSA key exchange can take place. In this case the session data are negotiated using the ephemeral/temporary RSA key and the RSA key supplied and certified by the certificate chain is only used for signing.

Under previous export restrictions, ciphers with RSA keys shorter (512 bits) than the usual key length of 1024 bits were created. To use these ciphers with RSA keys of usual length, an ephemeral key exchange must be performed, as the normal (certified) key cannot be directly used.

Using ephemeral RSA key exchange yields forward secrecy, as the connection can only be decrypted, when the RSA key is known. By generating a temporary RSA key inside the server application that is lost when the application is left, it becomes impossible for an attacker to decrypt past sessions, even if he gets hold of the normal (certified) RSA key, as this key was used for signing only. The downside is that creating a RSA key is computationally expensive.

Additionally, the use of ephemeral RSA key exchange is only allowed in the TLS standard, when the RSA key can be used for signing only, that is for export ciphers. Using ephemeral RSA key exchange for other purposes violates the standard and can break interoperability with clients. It is therefore strongly recommended to not use ephemeral RSA key exchange and use EDH (Ephemeral Diffie−Hellman) key exchange instead in order to achieve forward secrecy (see *SSL_CTX_set_tmp_dh_callback*(3)).

On OpenSSL servers ephemeral RSA key exchange is therefore disabled by default and must be explicitly enabled using the SSL_OP_EPHEMERAL_RSA option of *SSL_CTX_set_options*(3), violating the TLS/SSL standard. When ephemeral RSA key exchange is required for export ciphers, it will automatically be used without this option!

An application may either directly specify the key or can supply the key via a callback function. The callback approach has the advantage, that the callback may generate the key only in case it is actually needed. As the generation of a RSA key is however costly, it will lead to a significant delay in the handshake procedure. Another advantage of the callback function is that it can supply keys of different size (e.g. for SSL_OP_EPHEMERAL_RSA usage) while the explicit setting of the key is only useful for key size of 512 bits to satisfy the export restricted ciphers and does give away key length if a longer key would be allowed.

The **tmp_rsa_callback** is called with the **keylength** needed and the **is_export** information. The **is_export** flag is set, when the ephemeral RSA key exchange is performed with an export cipher.

EXAMPLES

Generate temporary RSA keys to prepare ephemeral RSA key exchange. As the generation of a RSA key costs a lot of computer time, they saved for later reuse. For demonstration purposes, two keys for 512 bits and 1024 bits respectively are generated.

```
...
/* Set up ephemeral RSA stuff */
RSA *rsa_512 = NULL;
RSA *rsa_1024 = NULL;

rsa_512 = RSA_generate_key(512,RSA_F4,NULL,NULL);
if (rsa_512 == NULL)
    evaluate_error_queue();

rsa_1024 = RSA_generate_key(1024,RSA_F4,NULL,NULL);
if (rsa_1024 == NULL)
  evaluate_error_queue();

...

RSA *tmp_rsa_callback(SSL *s, int is_export, int keylength)
{
   RSA *rsa_tmp=NULL;
```

```
        switch (keylength) {
        case 512:
          if (rsa_512)
            rsa_tmp = rsa_512;
          else { /* generate on the fly, should not happen in this example */
            rsa_tmp = RSA_generate_key(keylength,RSA_F4,NULL,NULL);
            rsa_512 = rsa_tmp; /* Remember for later reuse */
          }
          break;
        case 1024:
          if (rsa_1024)
            rsa_tmp=rsa_1024;
          else
            should_not_happen_in_this_example();
          break;
        default:
          /* Generating a key on the fly is very costly, so use what is there */
          if (rsa_1024)
            rsa_tmp=rsa_1024;
          else
            rsa_tmp=rsa_512; /* Use at least a shorter key */
        }
        return(rsa_tmp);
      }
```

RETURN VALUES

SSL_CTX_set_tmp_rsa_callback() and *SSL_set_tmp_rsa_callback()* do not return diagnostic output.

SSL_CTX_set_tmp_rsa() and *SSL_set_tmp_rsa()* do return 1 on success and 0 on failure. Check the error queue to find out the reason of failure.

SSL_CTX_need_tmp_rsa() and *SSL_need_tmp_rsa()* return 1 if a temporary RSA key is needed and 0 otherwise.

SEE ALSO

ssl (3), *SSL_CTX_set_cipher_list* (3), *SSL_CTX_set_options* (3), *SSL_CTX_set_tmp_dh_callback* (3), *SSL_new* (3), *openssl_ciphers* (1)

NAME

SSL_CTX_set_verify, SSL_set_verify, SSL_CTX_set_verify_depth, SSL_set_verify_depth − set peer certificate verification parameters

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_verify(SSL_CTX *ctx, int mode,
                        int (*verify_callback)(int, X509_STORE_CTX *));
void SSL_set_verify(SSL *s, int mode,
                    int (*verify_callback)(int, X509_STORE_CTX *));
void SSL_CTX_set_verify_depth(SSL_CTX *ctx,int depth);
void SSL_set_verify_depth(SSL *s, int depth);

int verify_callback(int preverify_ok, X509_STORE_CTX *x509_ctx);
```

DESCRIPTION

SSL_CTX_set_verify() sets the verification flags for **ctx** to be **mode** and specifies the **verify_callback** function to be used. If no callback function shall be specified, the NULL pointer can be used for **verify_callback**.

SSL_set_verify() sets the verification flags for **ssl** to be **mode** and specifies the **verify_callback** function to be used. If no callback function shall be specified, the NULL pointer can be used for **verify_callback**. In this case last **verify_callback** set specifically for this **ssl** remains. If no special **callback** was set before, the default callback for the underlying **ctx** is used, that was valid at the the time **ssl** was created with *SSL_new* (3).

SSL_CTX_set_verify_depth() sets the maximum **depth** for the certificate chain verification that shall be allowed for **ctx**. (See the BUGS section.)

SSL_set_verify_depth() sets the maximum **depth** for the certificate chain verification that shall be allowed for **ssl**. (See the BUGS section.)

NOTES

The verification of certificates can be controlled by a set of logically or'ed **mode** flags:

SSL_VERIFY_NONE

Server mode: the server will not send a client certificate request to the client, so the client will not send a certificate.

Client mode: if not using an anonymous cipher (by default disabled), the server will send a certificate which will be checked. The result of the certificate verification process can be checked after the TLS/SSL handshake using the *SSL_get_verify_result* (3) function. The handshake will be continued regardless of the verification result.

SSL_VERIFY_PEER

Server mode: the server sends a client certificate request to the client. The certificate returned (if any) is checked. If the verification process fails, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure. The behaviour can be controlled by the additional SSL_VERIFY_FAIL_IF_NO_PEER_CERT and SSL_VERIFY_CLIENT_ONCE flags.

Client mode: the server certificate is verified. If the verification process fails, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure. If no server certificate is sent, because an anonymous cipher is used, SSL_VERIFY_PEER is ignored.

SSL_VERIFY_FAIL_IF_NO_PEER_CERT

Server mode: if the client did not return a certificate, the TLS/SSL handshake is immediately terminated with a ''handshake failure'' alert. This flag must be used together with SSL_VERIFY_PEER.

Client mode: ignored

SSL_VERIFY_CLIENT_ONCE

Server mode: only request a client certificate on the initial TLS/SSL handshake. Do not ask for a client certificate again in case of a renegotiation. This flag must be used together with SSL_VERIFY_PEER.

Client mode: ignored

Exactly one of the **mode** flags SSL_VERIFY_NONE and SSL_VERIFY_PEER must be set at any time.

The actual verification procedure is performed either using the built-in verification procedure or using another application provided verification function set with *SSL_CTX_set_cert_verify_callback*(3). The following descriptions apply in the case of the built-in procedure. An application provided procedure also has access to the verify depth information and the *verify_callback()* function, but the way this information is used may be different.

SSL_CTX_set_verify_depth() and *SSL_set_verify_depth()* set the limit up to which depth certificates in a chain are used during the verification procedure. If the certificate chain is longer than allowed, the certificates above the limit are ignored. Error messages are generated as if these certificates would not be present, most likely a X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY will be issued. The depth count is ''level 0:peer certificate'', ''level 1: CA certificate'', ''level 2: higher level CA certificate'', and so on. Setting the maximum depth to 2 allows the levels 0, 1, and 2. The default depth limit is 9, allowing for the peer certificate and additional 9 CA certificates.

The **verify_callback** function is used to control the behaviour when the SSL_VERIFY_PEER flag is set. It must be supplied by the application and receives two arguments: **preverify_ok** indicates, whether the verification of the certificate in question was passed (preverify_ok=1) or not (preverify_ok=0). **x509_ctx** is a pointer to the complete context used for the certificate chain verification.

The certificate chain is checked starting with the deepest nesting level (the root CA certificate) and worked upward to the peer's certificate. At each level signatures and issuer attributes are checked. Whenever a verification error is found, the error number is stored in **x509_ctx** and **verify_callback** is called with **preverify_ok**=0. By applying X509_CTX_store_* functions **verify_callback** can locate the certificate in question and perform additional steps (see EXAMPLES). If no error is found for a certificate, **verify_callback** is called with **preverify_ok**=1 before advancing to the next level.

The return value of **verify_callback** controls the strategy of the further verification process. If **verify_callback** returns 0, the verification process is immediately stopped with ''verification failed'' state. If SSL_VERIFY_PEER is set, a verification failure alert is sent to the peer and the TLS/SSL handshake is terminated. If **verify_callback** returns 1, the verification process is continued. If **verify_callback** always returns 1, the TLS/SSL handshake will not be terminated with respect to verification failures and the connection will be established. The calling process can however retrieve the error code of the last verification error using *SSL_get_verify_result*(3) or by maintaining its own error storage managed by **verify_callback**.

If no **verify_callback** is specified, the default callback will be used. Its return value is identical to **preverify_ok**, so that any verification failure will lead to a termination of the TLS/SSL handshake with an alert message, if SSL_VERIFY_PEER is set.

BUGS

In client mode, it is not checked whether the SSL_VERIFY_PEER flag is set, but whether SSL_VERIFY_NONE is not set. This can lead to unexpected behaviour, if the SSL_VERIFY_PEER and SSL_VERIFY_NONE are not used as required (exactly one must be set at any time).

The certificate verification depth set with SSL[_CTX]_*verify_depth()* stops the verification at a certain depth. The error message produced will be that of an incomplete certificate chain and not X509_V_ERR_CERT_CHAIN_TOO_LONG as may be expected.

RETURN VALUES

The SSL*_set_verify*() functions do not provide diagnostic information.

EXAMPLES

The following code sequence realizes an example **verify_callback** function that will always continue the TLS/SSL handshake regardless of verification failure, if wished. The callback realizes a verification depth limit with more informational output.

All verification errors are printed, informations about the certificate chain are printed on request. The example is realized for a server that does allow but not require client certificates.

The example makes use of the ex_data technique to store application data into/retrieve application data from the SSL structure (see *SSL_get_ex_new_index* (3), *SSL_get_ex_data_X509_STORE_CTX_idx* (3)).

```
...
typedef struct {
  int verbose_mode;
  int verify_depth;
  int always_continue;
} mydata_t;
int mydata_index;
...
static int verify_callback(int preverify_ok, X509_STORE_CTX *ctx)
{
   char    buf[256];
   X509   *err_cert;
   int     err, depth;
   SSL    *ssl;
   mydata_t *mydata;

   err_cert = X509_STORE_CTX_get_current_cert(ctx);
   err = X509_STORE_CTX_get_error(ctx);
   depth = X509_STORE_CTX_get_error_depth(ctx);

   /*
    * Retrieve the pointer to the SSL of the connection currently treated
    * and the application specific data stored into the SSL object.
    */
   ssl = X509_STORE_CTX_get_ex_data(ctx, SSL_get_ex_data_X509_STORE_CTX_idx());
   mydata = SSL_get_ex_data(ssl, mydata_index);

   X509_NAME_oneline(X509_get_subject_name(err_cert), buf, 256);

   /*
    * Catch a too long certificate chain. The depth limit set using
    * SSL_CTX_set_verify_depth() is by purpose set to "limit+1" so
    * that whenever the "depth>verify_depth" condition is met, we
    * have violated the limit and want to log this error condition.
    * We must do it here, because the CHAIN_TOO_LONG error would not
    * be found explicitly; only errors introduced by cutting off the
    * additional certificates would be logged.
    */
   if (depth > mydata->verify_depth) {
       preverify_ok = 0;
       err = X509_V_ERR_CERT_CHAIN_TOO_LONG;
       X509_STORE_CTX_set_error(ctx, err);
   }
   if (!preverify_ok) {
       printf("verify error:num=%d:%s:depth=%d:%s\n", err,
                X509_verify_cert_error_string(err), depth, buf);
   }
   else if (mydata->verbose_mode)
   {
       printf("depth=%d:%s\n", depth, buf);
   }
```

```
    /*
     * At this point, err contains the last verification error. We can use
     * it for something special
     */
    if (!preverify_ok && (err == X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT))
    {
      X509_NAME_oneline(X509_get_issuer_name(ctx->current_cert), buf, 256);
      printf("issuer= %s\n", buf);
    }

    if (mydata->always_continue)
      return 1;
    else
      return preverify_ok;
}
...
mydata_t mydata;

...
mydata_index = SSL_get_ex_new_index(0, "mydata index", NULL, NULL, NULL);

...
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER|SSL_VERIFY_CLIENT_ONCE,
                   verify_callback);

/*
 * Let the verify_callback catch the verify_depth error so that we get
 * an appropriate error in the logfile.
 */
SSL_CTX_set_verify_depth(verify_depth + 1);

/*
 * Set up the SSL specific data into "mydata" and store it into th SSL
 * structure.
 */
mydata.verify_depth = verify_depth; ...
SSL_set_ex_data(ssl, mydata_index, &mydata);

...
SSL_accept(ssl);       /* check of success left out for clarity */
if (peer = SSL_get_peer_certificate(ssl))
{
  if (SSL_get_verify_result(ssl) == X509_V_OK)
  {
    /* The client sent a certificate which verified OK */
  }
}
```

SEE ALSO

ssl (3), *SSL_new* (3), *SSL_CTX_get_verify_mode* (3), *SSL_get_verify_result* (3), *SSL_CTX_load_verify_locations* (3), *SSL_get_peer_certificate* (3), *SSL_CTX_set_cert_verify_callback* (3), *SSL_get_ex_data_X509_STORE_CTX_idx* (3), *SSL_get_ex_new_index* (3)

NAME

SSL_CTX_use_certificate, SSL_CTX_use_certificate_ASN1, SSL_CTX_use_certificate_file, SSL_use_certificate, SSL_use_certificate_ASN1, SSL_use_certificate_file, SSL_CTX_use_certificate_chain_file, SSL_CTX_use_PrivateKey, SSL_CTX_use_PrivateKey_ASN1, SSL_CTX_use_PrivateKey_file, SSL_CTX_use_RSAPrivateKey, SSL_CTX_use_RSAPrivateKey_ASN1, SSL_CTX_use_RSAPrivateKey_file, SSL_use_PrivateKey_file, SSL_use_PrivateKey_ASN1, SSL_use_PrivateKey, SSL_use_RSAPrivateKey, SSL_use_RSAPrivateKey_ASN1, SSL_use_RSAPrivateKey_file, SSL_CTX_check_private_key, SSL_check_private_key − load certificate and key data

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_CTX_use_certificate(SSL_CTX *ctx, X509 *x);
int SSL_CTX_use_certificate_ASN1(SSL_CTX *ctx, int len, unsigned char *d);
int SSL_CTX_use_certificate_file(SSL_CTX *ctx, const char *file, int type);
int SSL_use_certificate(SSL *ssl, X509 *x);
int SSL_use_certificate_ASN1(SSL *ssl, unsigned char *d, int len);
int SSL_use_certificate_file(SSL *ssl, const char *file, int type);

int SSL_CTX_use_certificate_chain_file(SSL_CTX *ctx, const char *file);

int SSL_CTX_use_PrivateKey(SSL_CTX *ctx, EVP_PKEY *pkey);
int SSL_CTX_use_PrivateKey_ASN1(int pk, SSL_CTX *ctx, unsigned char *d,
                                long len);
int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx, const char *file, int type);
int SSL_CTX_use_RSAPrivateKey(SSL_CTX *ctx, RSA *rsa);
int SSL_CTX_use_RSAPrivateKey_ASN1(SSL_CTX *ctx, unsigned char *d, long len);
int SSL_CTX_use_RSAPrivateKey_file(SSL_CTX *ctx, const char *file, int type);
int SSL_use_PrivateKey(SSL *ssl, EVP_PKEY *pkey);
int SSL_use_PrivateKey_ASN1(int pk,SSL *ssl, unsigned char *d, long len);
int SSL_use_PrivateKey_file(SSL *ssl, const char *file, int type);
int SSL_use_RSAPrivateKey(SSL *ssl, RSA *rsa);
int SSL_use_RSAPrivateKey_ASN1(SSL *ssl, unsigned char *d, long len);
int SSL_use_RSAPrivateKey_file(SSL *ssl, const char *file, int type);

int SSL_CTX_check_private_key(const SSL_CTX *ctx);
int SSL_check_private_key(const SSL *ssl);
```

DESCRIPTION

These functions load the certificates and private keys into the SSL_CTX or SSL object, respectively.

The SSL_CTX_* class of functions loads the certificates and keys into the SSL_CTX object **ctx**. The information is passed to SSL objects **ssl** created from **ctx** with *SSL_new* (3) by copying, so that changes applied to **ctx** do not propagate to already existing SSL objects.

The SSL_* class of functions only loads certificates and keys into a specific SSL object. The specific information is kept, when *SSL_clear* (3) is called for this SSL object.

SSL_CTX_use_certificate() loads the certificate **x** into **ctx**, *SSL_use_certificate()* loads **x** into **ssl**. The rest of the certificates needed to form the complete certificate chain can be specified using the *SSL_CTX_add_extra_chain_cert* (3) function.

SSL_CTX_use_certificate_ASN1() loads the ASN1 encoded certificate from the memory location **d** (with length **len**) into **ctx**, *SSL_use_certificate_ASN1()* loads the ASN1 encoded certificate into **ssl**.

SSL_CTX_use_certificate_file() loads the first certificate stored in **file** into **ctx**. The formatting **type** of the certificate must be specified from the known types SSL_FILETYPE_PEM, SSL_FILETYPE_ASN1. *SSL_use_certificate_file()* loads the certificate from **file** into **ssl**. See the NOTES section on why *SSL_CTX_use_certificate_chain_file()* should be preferred.

SSL_CTX_use_certificate_chain_file() loads a certificate chain from **file** into **ctx**. The certificates must be in PEM format and must be sorted starting with the subject's certificate (actual client or server certificate), followed by intermediate CA certificates if applicable, and ending at the highest level (root) CA. There is no corresponding function working on a single SSL object.

SSL_CTX_use_PrivateKey() adds **pkey** as private key to **ctx**. *SSL_CTX_use_RSAPrivateKey()* adds the private key **rsa** of type RSA to **ctx**. *SSL_use_PrivateKey()* adds **pkey** as private key to **ssl**; *SSL_use_RSAPrivateKey()* adds **rsa** as private key of type RSA to **ssl**. If a certificate has already been set and the private does not belong to the certificate an error is returned. To change a certificate, private key pair the new certificate needs to be set with *SSL_use_certificate()* or *SSL_CTX_use_certificate()* before setting the private key with *SSL_CTX_use_PrivateKey()* or *SSL_use_PrivateKey()*.

SSL_CTX_use_PrivateKey_ASN1() adds the private key of type **pk** stored at memory location **d** (length **len**) to **ctx**. *SSL_CTX_use_RSAPrivateKey_ASN1()* adds the private key of type RSA stored at memory location **d** (length **len**) to **ctx**. *SSL_use_PrivateKey_ASN1()* and *SSL_use_RSAPrivateKey_ASN1()* add the private key to **ssl**.

SSL_CTX_use_PrivateKey_file() adds the first private key found in **file** to **ctx**. The formatting **type** of the certificate must be specified from the known types SSL_FILETYPE_PEM, SSL_FILETYPE_ASN1. *SSL_CTX_use_RSAPrivateKey_file()* adds the first private RSA key found in **file** to **ctx**. *SSL_use_PrivateKey_file()* adds the first private key found in **file** to **ssl**; *SSL_use_RSAPrivateKey_file()* adds the first private RSA key found to **ssl**.

SSL_CTX_check_private_key() checks the consistency of a private key with the corresponding certificate loaded into **ctx**. If more than one key/certificate pair (RSA/DSA) is installed, the last item installed will be checked. If e.g. the last item was a RSA certificate or key, the RSA key/certificate pair will be checked. *SSL_check_private_key()* performs the same check for **ssl**. If no key/certificate was explicitly added for this **ssl**, the last item added into **ctx** will be checked.

NOTES

The internal certificate store of OpenSSL can hold two private key/certificate pairs at a time: one key/certificate of type RSA and one key/certificate of type DSA. The certificate used depends on the cipher select, see also *SSL_CTX_set_cipher_list*(3).

When reading certificates and private keys from file, files of type SSL_FILETYPE_ASN1 (also known as **DER**, binary encoding) can only contain one certificate or private key, consequently *SSL_CTX_use_certificate_chain_file()* is only applicable to PEM formatting. Files of type SSL_FILETYPE_PEM can contain more than one item.

SSL_CTX_use_certificate_chain_file() adds the first certificate found in the file to the certificate store. The other certificates are added to the store of chain certificates using *SSL_CTX_add_extra_chain_cert*(3). There exists only one extra chain store, so that the same chain is appended to both types of certificates, RSA and DSA! If it is not intended to use both type of certificate at the same time, it is recommended to use the *SSL_CTX_use_certificate_chain_file()* instead of the *SSL_CTX_use_certificate_file()* function in order to allow the use of complete certificate chains even when no trusted CA storage is used or when the CA issuing the certificate shall not be added to the trusted CA storage.

If additional certificates are needed to complete the chain during the TLS negotiation, CA certificates are additionally looked up in the locations of trusted CA certificates, see *SSL_CTX_load_verify_locations*(3).

The private keys loaded from file can be encrypted. In order to successfully load encrypted keys, a function returning the passphrase must have been supplied, see *SSL_CTX_set_default_passwd_cb*(3). (Certificate files might be encrypted as well from the technical point of view, it however does not make sense as the data in the certificate is considered public anyway.)

RETURN VALUES

On success, the functions return 1. Otherwise check out the error stack to find out the reason.

SEE ALSO

ssl(3), *SSL_new*(3), *SSL_clear*(3), *SSL_CTX_load_verify_locations*(3), *SSL_CTX_set_default_passwd_cb*(3), *SSL_CTX_set_cipher_list*(3), *SSL_CTX_set_client_cert_cb*(3),

SSL_CTX_add_extra_chain_cert (3)

HISTORY

Support for DER encoded private keys (SSL_FILETYPE_ASN1) in *SSL_CTX_use_PrivateKey_file()* and *SSL_use_PrivateKey_file()* was added in 0.9.8 .

NAME

SSL_SESSION_free − free an allocated SSL_SESSION structure

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_SESSION_free(SSL_SESSION *session);
```

DESCRIPTION

SSL_SESSION_free() decrements the reference count of **session** and removes the **SSL_SESSION** structure pointed to by **session** and frees up the allocated memory, if the the reference count has reached 0.

NOTES

SSL_SESSION objects are allocated, when a TLS/SSL handshake operation is successfully completed. Depending on the settings, see *SSL_CTX_set_session_cache_mode* (3), the SSL_SESSION objects are internally referenced by the SSL_CTX and linked into its session cache. SSL objects may be using the SSL_SESSION object; as a session may be reused, several SSL objects may be using one SSL_SESSION object at the same time. It is therefore crucial to keep the reference count (usage information) correct and not delete a SSL_SESSION object that is still used, as this may lead to program failures due to dangling pointers. These failures may also appear delayed, e.g. when an SSL_SESSION object was completely freed as the reference count incorrectly became 0, but it is still referenced in the internal session cache and the cache list is processed during a *SSL_CTX_flush_sessions* (3) operation.

SSL_SESSION_free() must only be called for SSL_SESSION objects, for which the reference count was explicitly incremented (e.g. by calling *SSL_get1_session()*, see *SSL_get_session* (3)) or when the SSL_SESSION object was generated outside a TLS handshake operation, e.g. by using *d2i_SSL_SESSION* (3). It must not be called on other SSL_SESSION objects, as this would cause incorrect reference counts and therefore program failures.

RETURN VALUES

SSL_SESSION_free() does not provide diagnostic information.

SEE ALSO

ssl (3), *SSL_get_session* (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_CTX_flush_sessions* (3), *d2i_SSL_SESSION* (3)

NAME

SSL_SESSION_get_ex_new_index, SSL_SESSION_set_ex_data, SSL_SESSION_get_ex_data − internal application specific data functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_SESSION_get_ex_new_index(long argl, void *argp,
                CRYPTO_EX_new *new_func,
                CRYPTO_EX_dup *dup_func,
                CRYPTO_EX_free *free_func);

int SSL_SESSION_set_ex_data(SSL_SESSION *session, int idx, void *arg);

void *SSL_SESSION_get_ex_data(const SSL_SESSION *session, int idx);

typedef int new_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
                int idx, long argl, void *argp);
typedef void free_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
                int idx, long argl, void *argp);
typedef int dup_func(CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d,
                int idx, long argl, void *argp);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

SSL_SESSION_get_ex_new_index() is used to register a new index for application specific data.

SSL_SESSION_set_ex_data() is used to store application data at **arg** for **idx** into the **session** object.

SSL_SESSION_get_ex_data() is used to retrieve the information for **idx** from **session**.

A detailed description for the ***_get_ex_new_index()*** functionality can be found in *RSA_get_ex_new_index* (3). The ***_get_ex_data()*** and ***_set_ex_data()*** functionality is described in *CRYPTO_set_ex_data* (3).

WARNINGS

The application data is only maintained for sessions held in memory. The application data is not included when dumping the session with *i2d_SSL_SESSION()* (and all functions indirectly calling the dump functions like *PEM_write_SSL_SESSION()* and *PEM_write_bio_SSL_SESSION()*) and can therefore not be restored.

SEE ALSO

ssl (3), *RSA_get_ex_new_index* (3), *CRYPTO_set_ex_data* (3)

NAME

SSL_SESSION_get_time, SSL_SESSION_set_time, SSL_SESSION_get_timeout, SSL_SESSION_set_timeout − retrieve and manipulate session time and timeout settings

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_SESSION_get_time(const SSL_SESSION *s);
long SSL_SESSION_set_time(SSL_SESSION *s, long tm);
long SSL_SESSION_get_timeout(const SSL_SESSION *s);
long SSL_SESSION_set_timeout(SSL_SESSION *s, long tm);

long SSL_get_time(const SSL_SESSION *s);
long SSL_set_time(SSL_SESSION *s, long tm);
long SSL_get_timeout(const SSL_SESSION *s);
long SSL_set_timeout(SSL_SESSION *s, long tm);
```

DESCRIPTION

SSL_SESSION_get_time() returns the time at which the session **s** was established. The time is given in seconds since the Epoch and therefore compatible to the time delivered by the *time()* call.

SSL_SESSION_set_time() replaces the creation time of the session **s** with the chosen value **tm**.

SSL_SESSION_get_timeout() returns the timeout value set for session **s** in seconds.

SSL_SESSION_set_timeout() sets the timeout value for session **s** in seconds to **tm**.

The *SSL_get_time()*, *SSL_set_time()*, *SSL_get_timeout()*, and *SSL_set_timeout()* functions are synonyms for the SSL_SESSION_*() counterparts.

NOTES

Sessions are expired by examining the creation time and the timeout value. Both are set at creation time of the session to the actual time and the default timeout value at creation, respectively, as set by *SSL_CTX_set_timeout* (3). Using these functions it is possible to extend or shorten the lifetime of the session.

RETURN VALUES

SSL_SESSION_get_time() and *SSL_SESSION_get_timeout()* return the currently valid values.

SSL_SESSION_set_time() and *SSL_SESSION_set_timeout()* return 1 on success.

If any of the function is passed the NULL pointer for the session **s**, 0 is returned.

SEE ALSO

ssl (3), *SSL_CTX_set_timeout* (3), *SSL_get_default_timeout* (3)

NAME

SSL_accept − wait for a TLS/SSL client to initiate a TLS/SSL handshake

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_accept(SSL *ssl);
```

DESCRIPTION

SSL_accept() waits for a TLS/SSL client to initiate the TLS/SSL handshake. The communication channel must already have been set and assigned to the **ssl** by setting an underlying **BIO**.

NOTES

The behaviour of *SSL_accept()* depends on the underlying BIO.

If the underlying BIO is **blocking**, *SSL_accept()* will only return once the handshake has been finished or an error occurred, except for SGC (Server Gated Cryptography). For SGC, *SSL_accept()* may return with −1, but *SSL_get_error()* will yield **SSL_ERROR_WANT_READ/WRITE** and *SSL_accept()* should be called again.

If the underlying BIO is **non-blocking**, *SSL_accept()* will also return when the underlying BIO could not satisfy the needs of *SSL_accept()* to continue the handshake, indicating the problem by the return value −1. In this case a call to *SSL_get_error()* with the return value of *SSL_accept()* will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process then must repeat the call after taking appropriate action to satisfy the needs of *SSL_accept()*. The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but *select()* can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

RETURN VALUES

The following return values can occur:

1 The TLS/SSL handshake was successfully completed, a TLS/SSL connection has been established.

0 The TLS/SSL handshake was not successful but was shut down controlled and by the specifications of the TLS/SSL protocol. Call *SSL_get_error()* with the return value **ret** to find out the reason.

<0 The TLS/SSL handshake was not successful because a fatal error occurred either at the protocol level or a connection failure occurred. The shutdown was not clean. It can also occur of action is need to continue the operation for non-blocking BIOs. Call *SSL_get_error()* with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error (3), *SSL_connect* (3), *SSL_shutdown* (3), *ssl* (3), *openssl_bio* (3), *SSL_set_connect_state* (3), *SSL_do_handshake* (3), *SSL_CTX_new* (3)

NAME

SSL_alert_type_string, SSL_alert_type_string_long, SSL_alert_desc_string, SSL_alert_desc_string_long − get textual description of alert information

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

const char *SSL_alert_type_string(int value);
const char *SSL_alert_type_string_long(int value);

const char *SSL_alert_desc_string(int value);
const char *SSL_alert_desc_string_long(int value);
```

DESCRIPTION

SSL_alert_type_string() returns a one letter string indicating the type of the alert specified by **value**.

SSL_alert_type_string_long() returns a string indicating the type of the alert specified by **value**.

SSL_alert_desc_string() returns a two letter string as a short form describing the reason of the alert specified by **value**.

SSL_alert_desc_string_long() returns a string describing the reason of the alert specified by **value**.

NOTES

When one side of an SSL/TLS communication wants to inform the peer about a special situation, it sends an alert. The alert is sent as a special message and does not influence the normal data stream (unless its contents results in the communication being canceled).

A warning alert is sent, when a non-fatal error condition occurs. The ‘‘close notify’’ alert is sent as a warning alert. Other examples for non-fatal errors are certificate errors (‘‘certificate expired’’, ‘‘unsupported certificate’’), for which a warning alert may be sent. (The sending party may however decide to send a fatal error.) The receiving side may cancel the connection on reception of a warning alert on it discretion.

Several alert messages must be sent as fatal alert messages as specified by the TLS RFC. A fatal alert always leads to a connection abort.

RETURN VALUES

The following strings can occur for *SSL_alert_type_string()* or *SSL_alert_type_string_long()*:

‘‘W’’/‘‘warning’’
‘‘F’’/‘‘fatal’’
‘‘U’’/‘‘unknown’’
: This indicates that no support is available for this alert type. Probably **value** does not contain a correct alert message.

The following strings can occur for *SSL_alert_desc_string()* or *SSL_alert_desc_string_long()*:

‘‘CN’’/‘‘close notify’’
: The connection shall be closed. This is a warning alert.

‘‘UM’’/‘‘unexpected message’’
: An inappropriate message was received. This alert is always fatal and should never be observed in communication between proper implementations.

‘‘BM’’/‘‘bad record mac’’
: This alert is returned if a record is received with an incorrect MAC. This message is always fatal.

‘‘DF’’/‘‘decompression failure’’
: The decompression function received improper input (e.g. data that would expand to excessive length). This message is always fatal.

''HF''/''handshake failure''
Reception of a handshake_failure alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available. This is a fatal error.

''NC''/''no certificate''
A client, that was asked to send a certificate, does not send a certificate (SSLv3 only).

''BC''/''bad certificate''
A certificate was corrupt, contained signatures that did not verify correctly, etc

''UC''/''unsupported certificate''
A certificate was of an unsupported type.

''CR''/''certificate revoked''
A certificate was revoked by its signer.

''CE''/''certificate expired''
A certificate has expired or is not currently valid.

''CU''/''certificate unknown''
Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

''IP''/''illegal parameter''
A field in the handshake was out of range or inconsistent with other fields. This is always fatal.

''DC''/''decryption failed''
A TLSCiphertext decrypted in an invalid way: either it wasn't an even multiple of the block length or its padding values, when checked, weren't correct. This message is always fatal.

''RO''/''record overflow''
A TLSCiphertext record was received which had a length more than 2^14+2048 bytes, or a record decrypted to a TLSCompressed record with more than 2^14+1024 bytes. This message is always fatal.

''CA''/''unknown CA''
A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or couldn't be matched with a known, trusted CA. This message is always fatal.

''AD''/''access denied''
A valid certificate was received, but when access control was applied, the sender decided not to proceed with negotiation. This message is always fatal.

''DE''/''decode error''
A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This message is always fatal.

''CY''/''decrypt error''
A handshake cryptographic operation failed, including being unable to correctly verify a signature, decrypt a key exchange, or validate a finished message.

''ER''/''export restriction''
A negotiation not in compliance with export restrictions was detected; for example, attempting to transfer a 1024 bit ephemeral RSA key for the RSA_EXPORT handshake method. This message is always fatal.

''PV''/''protocol version''
The protocol version the client has attempted to negotiate is recognized, but not supported. (For example, old protocol versions might be avoided for security reasons). This message is always fatal.

''IS''/''insufficient security''
Returned instead of handshake_failure when a negotiation has failed specifically because the server requires ciphers more secure than those supported by the client. This message is always fatal.

‘‘IE’’/‘‘internal error’’
: An internal error unrelated to the peer or the correctness of the protocol makes it impossible to continue (such as a memory allocation failure). This message is always fatal.

‘‘US’’/‘‘user canceled’’
: This handshake is being canceled for some reason unrelated to a protocol failure. If the user cancels an operation after the handshake is complete, just closing the connection by sending a close_notify is more appropriate. This alert should be followed by a close_notify. This message is generally a warning.

‘‘NR’’/‘‘no renegotiation’’
: Sent by the client in response to a hello request or by the server in response to a client hello after initial handshaking. Either of these would normally lead to renegotiation; when that is not appropriate, the recipient should respond with this alert; at that point, the original requester can decide whether to proceed with the connection. One case where this would be appropriate would be where a server has spawned a process to satisfy a request; the process might receive security parameters (key length, authentication, etc.) at startup and it might be difficult to communicate changes to these parameters after that point. This message is always a warning.

‘‘UK’’/‘‘unknown’’
: This indicates that no description is available for this alert type. Probably **value** does not contain a correct alert message.

SEE ALSO

ssl (3), *SSL_CTX_set_info_callback* (3)

NAME

SSL_clear − reset SSL object to allow another connection

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_clear(SSL *ssl);
```

DESCRIPTION

Reset **ssl** to allow another connection. All settings (method, ciphers, BIOs) are kept.

NOTES

SSL_clear is used to prepare an SSL object for a new connection. While all settings are kept, a side effect is the handling of the current SSL session. If a session is still **open**, it is considered bad and will be removed from the session cache, as required by RFC2246. A session is considered open, if *SSL_shutdown*(3) was not called for the connection or at least *SSL_set_shutdown*(3) was used to set the SSL_SENT_SHUTDOWN state.

If a session was closed cleanly, the session object will be kept and all settings corresponding. This explicitly means, that e.g. the special method used during the session will be kept for the next handshake. So if the session was a TLSv1 session, a SSL client object will use a TLSv1 client method for the next handshake and a SSL server object will use a TLSv1 server method, even if SSLv23_*_methods were chosen on startup. This will might lead to connection failures (see *SSL_new*(3)) for a description of the method's properties.

WARNINGS

SSL_clear() resets the SSL object to allow for another connection. The reset operation however keeps several settings of the last sessions (some of these settings were made automatically during the last handshake). It only makes sense when opening a new session (or reusing an old one) with the same peer that shares these settings. *SSL_clear()* is not a short form for the sequence *SSL_free*(3); *SSL_new*(3); .

RETURN VALUES

The following return values can occur:

0 The *SSL_clear()* operation could not be performed. Check the error stack to find out the reason.

1 The *SSL_clear()* operation was successful.

SSL_new(3), *SSL_free*(3), *SSL_shutdown*(3), *SSL_set_shutdown*(3), *SSL_CTX_set_options*(3), *ssl*(3), *SSL_CTX_set_client_cert_cb*(3)

NAME

SSL_connect − initiate the TLS/SSL handshake with an TLS/SSL server

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_connect(SSL *ssl);
```

DESCRIPTION

SSL_connect() initiates the TLS/SSL handshake with a server. The communication channel must already have been set and assigned to the **ssl** by setting an underlying **BIO**.

NOTES

The behaviour of *SSL_connect()* depends on the underlying BIO.

If the underlying BIO is **blocking**, *SSL_connect()* will only return once the handshake has been finished or an error occurred.

If the underlying BIO is **non-blocking**, *SSL_connect()* will also return when the underlying BIO could not satisfy the needs of *SSL_connect()* to continue the handshake, indicating the problem by the return value −1. In this case a call to *SSL_get_error()* with the return value of *SSL_connect()* will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process then must repeat the call after taking appropriate action to satisfy the needs of *SSL_connect()*. The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but *select()* can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

RETURN VALUES

The following return values can occur:

1 The TLS/SSL handshake was successfully completed, a TLS/SSL connection has been established.

0 The TLS/SSL handshake was not successful but was shut down controlled and by the specifications of the TLS/SSL protocol. Call *SSL_get_error()* with the return value **ret** to find out the reason.

<0 The TLS/SSL handshake was not successful, because a fatal error occurred either at the protocol level or a connection failure occurred. The shutdown was not clean. It can also occur of action is need to continue the operation for non-blocking BIOs. Call *SSL_get_error()* with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error (3), *SSL_accept* (3), *SSL_shutdown* (3), *ssl* (3), *openssl_bio* (3), *SSL_set_connect_state* (3), *SSL_do_handshake* (3), *SSL_CTX_new* (3)

NAME

SSL_do_handshake − perform a TLS/SSL handshake

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_do_handshake(SSL *ssl);
```

DESCRIPTION

SSL_do_handshake() will wait for a SSL/TLS handshake to take place. If the connection is in client mode, the handshake will be started. The handshake routines may have to be explicitly set in advance using either *SSL_set_connect_state* (3) or *SSL_set_accept_state* (3).

NOTES

The behaviour of *SSL_do_handshake()* depends on the underlying BIO.

If the underlying BIO is **blocking**, *SSL_do_handshake()* will only return once the handshake has been finished or an error occurred, except for SGC (Server Gated Cryptography). For SGC, *SSL_do_handshake()* may return with −1, but *SSL_get_error()* will yield **SSL_ERROR_WANT_READ/WRITE** and *SSL_do_handshake()* should be called again.

If the underlying BIO is **non-blocking**, *SSL_do_handshake()* will also return when the underlying BIO could not satisfy the needs of *SSL_do_handshake()* to continue the handshake. In this case a call to *SSL_get_error()* with the return value of *SSL_do_handshake()* will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process then must repeat the call after taking appropriate action to satisfy the needs of *SSL_do_handshake()*. The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but *select()* can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

RETURN VALUES

The following return values can occur:

1 The TLS/SSL handshake was successfully completed, a TLS/SSL connection has been established.

0 The TLS/SSL handshake was not successful but was shut down controlled and by the specifications of the TLS/SSL protocol. Call *SSL_get_error()* with the return value **ret** to find out the reason.

<0 The TLS/SSL handshake was not successful because a fatal error occurred either at the protocol level or a connection failure occurred. The shutdown was not clean. It can also occur of action is need to continue the operation for non-blocking BIOs. Call *SSL_get_error()* with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error (3), *SSL_connect* (3), *SSL_accept* (3), *ssl* (3), *openssl_bio* (3), *SSL_set_connect_state* (3)

NAME

SSL_free − free an allocated SSL structure

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_free(SSL *ssl);
```

DESCRIPTION

SSL_free() decrements the reference count of **ssl**, and removes the SSL structure pointed to by **ssl** and frees up the allocated memory if the the reference count has reached 0.

NOTES

SSL_free() also calls the *free()*ing procedures for indirectly affected items, if applicable: the buffering BIO, the read and write BIOs, cipher lists specially created for this **ssl**, the **SSL_SESSION**. Do not explicitly free these indirectly freed up items before or after calling *SSL_free()*, as trying to free things twice may lead to program failure.

The ssl session has reference counts from two users: the SSL object, for which the reference count is removed by *SSL_free()* and the internal session cache. If the session is considered bad, because *SSL_shutdown* (3) was not called for the connection and *SSL_set_shutdown* (3) was not used to set the SSL_SENT_SHUTDOWN state, the session will also be removed from the session cache as required by RFC2246.

RETURN VALUES

SSL_free() does not provide diagnostic information.

SSL_new (3), *SSL_clear* (3), *SSL_shutdown* (3), *SSL_set_shutdown* (3), *ssl* (3)

NAME

SSL_get_SSL_CTX − get the SSL_CTX from which an SSL is created

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

SSL_CTX *SSL_get_SSL_CTX(const SSL *ssl);
```

DESCRIPTION

SSL_get_SSL_CTX() returns a pointer to the SSL_CTX object, from which **ssl** was created with *SSL_new*(3).

RETURN VALUES

The pointer to the SSL_CTX object is returned.

SEE ALSO

ssl(3), *SSL_new*(3)

NAME

SSL_get_ciphers, SSL_get_cipher_list − get list of available SSL_CIPHERs

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

STACK_OF(SSL_CIPHER) *SSL_get_ciphers(const SSL *ssl);
const char *SSL_get_cipher_list(const SSL *ssl, int priority);
```

DESCRIPTION

SSL_get_ciphers() returns the stack of available SSL_CIPHERs for **ssl**, sorted by preference. If **ssl** is NULL or no ciphers are available, NULL is returned.

SSL_get_cipher_list() returns a pointer to the name of the SSL_CIPHER listed for **ssl** with **priority**. If **ssl** is NULL, no ciphers are available, or there are less ciphers than **priority** available, NULL is returned.

NOTES

The details of the ciphers obtained by *SSL_get_ciphers()* can be obtained using the *SSL_CIPHER_get_name* (3) family of functions.

Call *SSL_get_cipher_list()* with **priority** starting from 0 to obtain the sorted list of available ciphers, until NULL is returned.

RETURN VALUES

See DESCRIPTION

SEE ALSO

ssl (3), *SSL_CTX_set_cipher_list* (3), *SSL_CIPHER_get_name* (3)

NAME

SSL_get_client_CA_list, SSL_CTX_get_client_CA_list − get list of client CAs

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

STACK_OF(X509_NAME) *SSL_get_client_CA_list(const SSL *s);
STACK_OF(X509_NAME) *SSL_CTX_get_client_CA_list(const SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_get_client_CA_list() returns the list of client CAs explicitly set for **ctx** using *SSL_CTX_set_client_CA_list*(3).

SSL_get_client_CA_list() returns the list of client CAs explicitly set for **ssl** using *SSL_set_client_CA_list()* or **ssl**'s SSL_CTX object with *SSL_CTX_set_client_CA_list*(3), when in server mode. In client mode, SSL_get_client_CA_list returns the list of client CAs sent from the server, if any.

RETURN VALUES

SSL_CTX_set_client_CA_list() and *SSL_set_client_CA_list()* do not return diagnostic information.

SSL_CTX_add_client_CA() and *SSL_add_client_CA()* have the following return values:

STACK_OF(X509_NAMES)

List of CA names explicitly set (for **ctx** or in server mode) or send by the server (client mode).

NULL

No client CA list was explicitly set (for **ctx** or in server mode) or the server did not send a list of CAs (client mode).

SEE ALSO

ssl(3), *SSL_CTX_set_client_CA_list*(3), *SSL_CTX_set_client_cert_cb*(3)

NAME

SSL_get_current_cipher, SSL_get_cipher, SSL_get_cipher_name, SSL_get_cipher_bits, SSL_get_cipher_version − get SSL_CIPHER of a connection

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

SSL_CIPHER *SSL_get_current_cipher(const SSL *ssl);
#define SSL_get_cipher(s) \
                SSL_CIPHER_get_name(SSL_get_current_cipher(s))
#define SSL_get_cipher_name(s) \
                SSL_CIPHER_get_name(SSL_get_current_cipher(s))
#define SSL_get_cipher_bits(s,np) \
                SSL_CIPHER_get_bits(SSL_get_current_cipher(s),np)
#define SSL_get_cipher_version(s) \
                SSL_CIPHER_get_version(SSL_get_current_cipher(s))
```

DESCRIPTION

SSL_get_current_cipher() returns a pointer to an SSL_CIPHER object containing the description of the actually used cipher of a connection established with the **ssl** object.

SSL_get_cipher() and *SSL_get_cipher_name()* are identical macros to obtain the name of the currently used cipher. *SSL_get_cipher_bits()* is a macro to obtain the number of secret/algorithm bits used and *SSL_get_cipher_version()* returns the protocol name. See *SSL_CIPHER_get_name* (3) for more details.

RETURN VALUES

SSL_get_current_cipher() returns the cipher actually used or NULL, when no session has been established.

SEE ALSO

ssl (3), *SSL_CIPHER_get_name* (3)

NAME

SSL_get_default_timeout − get default session timeout value

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_get_default_timeout(const SSL *ssl);
```

DESCRIPTION

SSL_get_default_timeout() returns the default timeout value assigned to SSL_SESSION objects negotiated for the protocol valid for **ssl**.

NOTES

Whenever a new session is negotiated, it is assigned a timeout value, after which it will not be accepted for session reuse. If the timeout value was not explicitly set using *SSL_CTX_set_timeout*(3), the hardcoded default timeout for the protocol will be used.

SSL_get_default_timeout() return this hardcoded value, which is 300 seconds for all currently supported protocols (SSLv2, SSLv3, and TLSv1).

RETURN VALUES

See description.

SEE ALSO

ssl(3), *SSL_CTX_set_session_cache_mode*(3), *SSL_SESSION_get_time*(3), *SSL_CTX_flush_sessions*(3), *SSL_get_default_timeout*(3)

NAME

SSL_get_error − obtain result code for TLS/SSL I/O operation

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_get_error(const SSL *ssl, int ret);
```

DESCRIPTION

SSL_get_error() returns a result code (suitable for the C ‘‘switch’’ statement) for a preceding call to *SSL_connect()*, *SSL_accept()*, *SSL_do_handshake()*, *SSL_read()*, *SSL_peek()*, or *SSL_write()* on **ssl**. The value returned by that TLS/SSL I/O function must be passed to *SSL_get_error()* in parameter **ret**.

In addition to **ssl** and **ret**, *SSL_get_error()* inspects the current thread’s OpenSSL error queue. Thus, *SSL_get_error()* must be used in the same thread that performed the TLS/SSL I/O operation, and no other OpenSSL function calls should appear in between. The current thread’s error queue must be empty before the TLS/SSL I/O operation is attempted, or *SSL_get_error()* will not work reliably.

RETURN VALUES

The following return values can currently occur:

SSL_ERROR_NONE

The TLS/SSL I/O operation completed. This result code is returned if and only if **ret > 0**.

SSL_ERROR_ZERO_RETURN

The TLS/SSL connection has been closed. If the protocol version is SSL 3.0 or TLS 1.0, this result code is returned only if a closure alert has occurred in the protocol, i.e. if the connection has been closed cleanly. Note that in this case **SSL_ERROR_ZERO_RETURN** does not necessarily indicate that the underlying transport has been closed.

SSL_ERROR_WANT_READ, SSL_ERROR_WANT_WRITE

The operation did not complete; the same TLS/SSL I/O function should be called again later. If, by then, the underlying **BIO** has data available for reading (if the result code is **SSL_ERROR_WANT_READ**) or allows writing data (**SSL_ERROR_WANT_WRITE**), then some TLS/SSL protocol progress will take place, i.e. at least part of an TLS/SSL record will be read or written. Note that the retry may again lead to a **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE** condition. There is no fixed upper limit for the number of iterations that may be necessary until progress becomes visible at application protocol level.

For socket **BIO**s (e.g. when *SSL_set_fd()* was used), *select()* or *poll()* on the underlying socket can be used to find out when the TLS/SSL I/O function should be retried.

Caveat: Any TLS/SSL I/O function can lead to either of **SSL_ERROR_WANT_READ** and **SSL_ERROR_WANT_WRITE**. In particular, *SSL_read()* or *SSL_peek()* may want to write data and *SSL_write()* may want to read data. This is mainly because TLS/SSL handshakes may occur at any time during the protocol (initiated by either the client or the server); *SSL_read()*, *SSL_peek()*, and *SSL_write()* will handle any pending handshakes.

SSL_ERROR_WANT_CONNECT, SSL_ERROR_WANT_ACCEPT

The operation did not complete; the same TLS/SSL I/O function should be called again later. The underlying BIO was not connected yet to the peer and the call would block in *connect()*/*accept()*. The SSL function should be called again when the connection is established. These messages can only appear with a *BIO_s_connect()* or *BIO_s_accept()* BIO, respectively. In order to find out, when the connection has been successfully established, on many platforms *select()* or *poll()* for writing on the socket file descriptor can be used.

SSL_ERROR_WANT_X509_LOOKUP

The operation did not complete because an application callback set by *SSL_CTX_set_client_cert_cb()* has asked to be called again. The TLS/SSL I/O function should be called again later. Details depend

on the application.

SSL_ERROR_SYSCALL
: Some I/O error occurred. The OpenSSL error queue may contain more information on the error. If the error queue is empty (i.e. *ERR_get_error()* returns 0), **ret** can be used to find out more about the error: If **ret == 0**, an EOF was observed that violates the protocol. If **ret == −1**, the underlying **BIO** reported an I/O error (for socket I/O on Unix systems, consult **errno** for details).

SSL_ERROR_SSL
: A failure in the SSL library occurred, usually a protocol error. The OpenSSL error queue contains more information on the error.

SEE ALSO

ssl (3), *openssl_err* (3)

HISTORY

SSL_get_error() was added in SSLeay 0.8.

NAME

SSL_get_ex_data_X509_STORE_CTX_idx − get ex_data index to access SSL structure from X509_STORE_CTX

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_get_ex_data_X509_STORE_CTX_idx(void);
```

DESCRIPTION

SSL_get_ex_data_X509_STORE_CTX_idx() returns the index number under which the pointer to the SSL object is stored into the X509_STORE_CTX object.

NOTES

Whenever a X509_STORE_CTX object is created for the verification of the peers certificate during a handshake, a pointer to the SSL object is stored into the X509_STORE_CTX object to identify the connection affected. To retrieve this pointer the *X509_STORE_CTX_get_ex_data()* function can be used with the correct index. This index is globally the same for all X509_STORE_CTX objects and can be retrieved using *SSL_get_ex_data_X509_STORE_CTX_idx()*. The index value is set when *SSL_get_ex_data_X509_STORE_CTX_idx()* is first called either by the application program directly or indirectly during other SSL setup functions or during the handshake.

The value depends on other index values defined for X509_STORE_CTX objects before the SSL index is created.

RETURN VALUES

>=0
: The index value to access the pointer.

<0 An error occurred, check the error stack for a detailed error message.

EXAMPLES

The index returned from *SSL_get_ex_data_X509_STORE_CTX_idx()* allows to access the SSL object for the connection to be accessed during the *verify_callback()* when checking the peers certificate. Please check the example in *SSL_CTX_set_verify* (3),

SEE ALSO

ssl (3), *SSL_CTX_set_verify* (3), *CRYPTO_set_ex_data* (3)

NAME

SSL_get_ex_new_index, SSL_set_ex_data, SSL_get_ex_data − internal application specific data functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_get_ex_new_index(long argl, void *argp,
                CRYPTO_EX_new *new_func,
                CRYPTO_EX_dup *dup_func,
                CRYPTO_EX_free *free_func);

int SSL_set_ex_data(SSL *ssl, int idx, void *arg);

void *SSL_get_ex_data(const SSL *ssl, int idx);

typedef int new_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
                int idx, long argl, void *argp);
typedef void free_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
                int idx, long argl, void *argp);
typedef int dup_func(CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d,
                int idx, long argl, void *argp);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

SSL_get_ex_new_index() is used to register a new index for application specific data.

SSL_set_ex_data() is used to store application data at **arg** for **idx** into the **ssl** object.

SSL_get_ex_data() is used to retrieve the information for **idx** from **ssl**.

A detailed description for the ***_get_ex_new_index()*** functionality can be found in *RSA_get_ex_new_index* (3). The ***_get_ex_data()*** and ***_set_ex_data()*** functionality is described in *CRYPTO_set_ex_data* (3).

EXAMPLES

An example on how to use the functionality is included in the example *verify_callback()* in *SSL_CTX_set_verify* (3).

SEE ALSO

ssl (3), *RSA_get_ex_new_index* (3), *CRYPTO_set_ex_data* (3), *SSL_CTX_set_verify* (3)

NAME

SSL_get_fd − get file descriptor linked to an SSL object

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_get_fd(const SSL *ssl);
int SSL_get_rfd(const SSL *ssl);
int SSL_get_wfd(const SSL *ssl);
```

DESCRIPTION

SSL_get_fd() returns the file descriptor which is linked to **ssl**. *SSL_get_rfd()* and *SSL_get_wfd()* return the file descriptors for the read or the write channel, which can be different. If the read and the write channel are different, *SSL_get_fd()* will return the file descriptor of the read channel.

RETURN VALUES

The following return values can occur:

−1 The operation failed, because the underlying BIO is not of the correct type (suitable for file descriptors).

>=0
The file descriptor linked to **ssl**.

SEE ALSO

SSL_set_fd (3), *ssl* (3) , *openssl_bio* (3)

NAME

SSL_get_peer_cert_chain − get the X509 certificate chain of the peer

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

STACKOF(X509) *SSL_get_peer_cert_chain(const SSL *ssl);
```

DESCRIPTION

SSL_get_peer_cert_chain() returns a pointer to STACKOF(X509) certificates forming the certificate chain of the peer. If called on the client side, the stack also contains the peer's certificate; if called on the server side, the peer's certificate must be obtained separately using *SSL_get_peer_certificate* (3). If the peer did not present a certificate, NULL is returned.

NOTES

The peer certificate chain is not necessarily available after reusing a session, in which case a NULL pointer is returned.

The reference count of the STACKOF(X509) object is not incremented. If the corresponding session is freed, the pointer must not be used any longer.

RETURN VALUES

The following return values can occur:

NULL
: No certificate was presented by the peer or no connection was established or the certificate chain is no longer available when a session is reused.

Pointer to a STACKOF(X509)
: The return value points to the certificate chain presented by the peer.

SEE ALSO

ssl (3), *SSL_get_peer_certificate* (3)

NAME

SSL_get_peer_certificate − get the X509 certificate of the peer

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

X509 *SSL_get_peer_certificate(const SSL *ssl);
```

DESCRIPTION

SSL_get_peer_certificate() returns a pointer to the X509 certificate the peer presented. If the peer did not present a certificate, NULL is returned.

NOTES

Due to the protocol definition, a TLS/SSL server will always send a certificate, if present. A client will only send a certificate when explicitly requested to do so by the server (see *SSL_CTX_set_verify* (3)). If an anonymous cipher is used, no certificates are sent.

That a certificate is returned does not indicate information about the verification state, use *SSL_get_verify_result* (3) to check the verification state.

The reference count of the X509 object is incremented by one, so that it will not be destroyed when the session containing the peer certificate is freed. The X509 object must be explicitly freed using *X509_free()*.

RETURN VALUES

The following return values can occur:

NULL
: No certificate was presented by the peer or no connection was established.

Pointer to an X509 certificate
: The return value points to the certificate presented by the peer.

SEE ALSO

ssl (3), *SSL_get_verify_result* (3), *SSL_CTX_set_verify* (3)

NAME

SSL_get_rbio − get BIO linked to an SSL object

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

BIO *SSL_get_rbio(SSL *ssl);
BIO *SSL_get_wbio(SSL *ssl);
```

DESCRIPTION

SSL_get_rbio() and *SSL_get_wbio()* return pointers to the BIOs for the read or the write channel, which can be different. The reference count of the BIO is not incremented.

RETURN VALUES

The following return values can occur:

NULL
: No BIO was connected to the SSL object

Any other pointer
: The BIO linked to **ssl**.

SEE ALSO

SSL_set_bio (3), *ssl* (3) , *openssl_bio* (3)

NAME

SSL_get_session − retrieve TLS/SSL session data

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

SSL_SESSION *SSL_get_session(const SSL *ssl);
SSL_SESSION *SSL_get0_session(const SSL *ssl);
SSL_SESSION *SSL_get1_session(SSL *ssl);
```

DESCRIPTION

SSL_get_session() returns a pointer to the **SSL_SESSION** actually used in **ssl**. The reference count of the **SSL_SESSION** is not incremented, so that the pointer can become invalid by other operations.

SSL_get0_session() is the same as *SSL_get_session()*.

SSL_get1_session() is the same as *SSL_get_session()*, but the reference count of the **SSL_SESSION** is incremented by one.

NOTES

The ssl session contains all information required to re-establish the connection without a new handshake.

SSL_get0_session() returns a pointer to the actual session. As the reference counter is not incremented, the pointer is only valid while the connection is in use. If *SSL_clear* (3) or *SSL_free* (3) is called, the session may be removed completely (if considered bad), and the pointer obtained will become invalid. Even if the session is valid, it can be removed at any time due to timeout during *SSL_CTX_flush_sessions* (3).

If the data is to be kept, *SSL_get1_session()* will increment the reference count, so that the session will not be implicitly removed by other operations but stays in memory. In order to remove the session *SSL_SESSION_free* (3) must be explicitly called once to decrement the reference count again.

SSL_SESSION objects keep internal link information about the session cache list, when being inserted into one SSL_CTX object's session cache. One SSL_SESSION object, regardless of its reference count, must therefore only be used with one SSL_CTX object (and the SSL objects created from this SSL_CTX object).

RETURN VALUES

The following return values can occur:

NULL
: There is no session available in **ssl**.

Pointer to an SSL
: The return value points to the data of an SSL session.

SEE ALSO

ssl (3), *SSL_free* (3), *SSL_clear* (3), *SSL_SESSION_free* (3)

NAME

SSL_get_verify_result − get result of peer certificate verification

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_get_verify_result(const SSL *ssl);
```

DESCRIPTION

SSL_get_verify_result() returns the result of the verification of the X509 certificate presented by the peer, if any.

NOTES

SSL_get_verify_result() can only return one error code while the verification of a certificate can fail because of many reasons at the same time. Only the last verification error that occurred during the processing is available from *SSL_get_verify_result()*.

The verification result is part of the established session and is restored when a session is reused.

BUGS

If no peer certificate was presented, the returned result code is X509_V_OK. This is because no verification error occurred, it does however not indicate success. *SSL_get_verify_result()* is only useful in connection with *SSL_get_peer_certificate* (3).

RETURN VALUES

The following return values can currently occur:

X509_V_OK
: The verification succeeded or no peer certificate was presented.

Any other value
: Documented in *openssl_verify* (1).

SEE ALSO

ssl (3), *SSL_set_verify_result* (3), *SSL_get_peer_certificate* (3), *openssl_verify* (1)

NAME

SSL_get_version − get the protocol version of a connection.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

const char *SSL_get_version(const SSL *ssl);
```

DESCRIPTION

SSL_get_cipher_version() returns the name of the protocol used for the connection **ssl**.

RETURN VALUES

The following strings can occur:

SSLv2
: The connection uses the SSLv2 protocol.

SSLv3
: The connection uses the SSLv3 protocol.

TLSv1
: The connection uses the TLSv1 protocol.

unknown
: This indicates that no version has been set (no connection established).

SEE ALSO

ssl (3)

NAME

SSL_library_init, OpenSSL_add_ssl_algorithms, SSLeay_add_ssl_algorithms − initialize SSL library by registering algorithms

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_library_init(void);
#define OpenSSL_add_ssl_algorithms()    SSL_library_init()
#define SSLeay_add_ssl_algorithms()     SSL_library_init()
```

DESCRIPTION

SSL_library_init() registers the available ciphers and digests.

OpenSSL_add_ssl_algorithms() and *SSLeay_add_ssl_algorithms()* are synonyms for *SSL_library_init()*.

NOTES

SSL_library_init() must be called before any other action takes place. *SSL_library_init()* is not reentrant.

WARNING

SSL_library_init() only registers ciphers. Another important initialization is the seeding of the PRNG (Pseudo Random Number Generator), which has to be performed separately.

EXAMPLES

A typical TLS/SSL application will start with the library initialization, will provide readable error messages and will seed the PRNG.

```
SSL_load_error_strings();               /* readable error messages */
SSL_library_init();                     /* initialize library */
actions_to_seed_PRNG();
```

RETURN VALUES

SSL_library_init() always returns ‘‘1’’, so it is safe to discard the return value.

SEE ALSO

ssl (3), *SSL_load_error_strings* (3), *RAND_add* (3)

NAME

SSL_load_client_CA_file − load certificate names from file

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

STACK_OF(X509_NAME) *SSL_load_client_CA_file(const char *file);
```

DESCRIPTION

SSL_load_client_CA_file() reads certificates from **file** and returns a STACK_OF(X509_NAME) with the subject names found.

NOTES

SSL_load_client_CA_file() reads a file of PEM formatted certificates and extracts the X509_NAMES of the certificates found. While the name suggests the specific usage as support function for *SSL_CTX_set_client_CA_list* (3), it is not limited to CA certificates.

EXAMPLES

Load names of CAs from file and use it as a client CA list:

```
SSL_CTX *ctx;
STACK_OF(X509_NAME) *cert_names;

...
cert_names = SSL_load_client_CA_file("/path/to/CAfile.pem");
if (cert_names != NULL)
  SSL_CTX_set_client_CA_list(ctx, cert_names);
else
  error_handling();
...
```

RETURN VALUES

The following return values can occur:

NULL
: The operation failed, check out the error stack for the reason.

Pointer to STACK_OF(X509_NAME)
: Pointer to the subject names of the successfully read certificates.

SEE ALSO

ssl (3), *SSL_CTX_set_client_CA_list* (3)

NAME

SSL_new − create a new SSL structure for a connection

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

SSL *SSL_new(SSL_CTX *ctx);
```

DESCRIPTION

SSL_new() creates a new **SSL** structure which is needed to hold the data for a TLS/SSL connection. The new structure inherits the settings of the underlying context **ctx**: connection method (SSLv2/v3/TLSv1), options, verification settings, timeout settings.

RETURN VALUES

The following return values can occur:

NULL
: The creation of a new SSL structure failed. Check the error stack to find out the reason.

Pointer to an SSL structure
: The return value points to an allocated SSL structure.

SEE ALSO

SSL_free (3), *SSL_clear* (3), *SSL_CTX_set_options* (3), *SSL_get_SSL_CTX* (3), *ssl* (3)

NAME

SSL_pending − obtain number of readable bytes buffered in an SSL object

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_pending(const SSL *ssl);
```

DESCRIPTION

SSL_pending() returns the number of bytes which are available inside **ssl** for immediate read.

NOTES

Data are received in blocks from the peer. Therefore data can be buffered inside **ssl** and are ready for immediate retrieval with *SSL_read* (3).

RETURN VALUES

The number of bytes pending is returned.

BUGS

SSL_pending() takes into account only bytes from the TLS/SSL record that is currently being processed (if any). If the **SSL** object's *read_ahead* flag is set, additional protocol bytes may have been read containing more TLS/SSL records; these are ignored by *SSL_pending()*.

Up to OpenSSL 0.9.6, *SSL_pending()* does not check if the record type of pending data is application data.

SEE ALSO

SSL_read (3), *ssl* (3)

NAME

SSL_read − read bytes from a TLS/SSL connection.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_read(SSL *ssl, void *buf, int num);
```

DESCRIPTION

SSL_read() tries to read **num** bytes from the specified **ssl** into the buffer **buf**.

NOTES

If necessary, *SSL_read()* will negotiate a TLS/SSL session, if not already explicitly performed by *SSL_connect* (3) or *SSL_accept* (3). If the peer requests a re−negotiation, it will be performed transparently during the *SSL_read()* operation. The behaviour of *SSL_read()* depends on the underlying BIO.

For the transparent negotiation to succeed, the **ssl** must have been initialized to client or server mode. This is being done by calling *SSL_set_connect_state* (3) or *SSL_set_accept_state()* before the first call to an *SSL_read()* or *SSL_write* (3) function.

SSL_read() works based on the SSL/TLS records. The data are received in records (with a maximum record size of 16kB for SSLv3/TLSv1). Only when a record has been completely received, it can be processed (decryption and check of integrity). Therefore data that was not retrieved at the last call of *SSL_read()* can still be buffered inside the SSL layer and will be retrieved on the next call to *SSL_read()*. If **num** is higher than the number of bytes buffered, *SSL_read()* will return with the bytes buffered. If no more bytes are in the buffer, *SSL_read()* will trigger the processing of the next record. Only when the record has been received and processed completely, *SSL_read()* will return reporting success. At most the contents of the record will be returned. As the size of an SSL/TLS record may exceed the maximum packet size of the underlying transport (e.g. TCP), it may be necessary to read several packets from the transport layer before the record is complete and *SSL_read()* can succeed.

If the underlying BIO is **blocking**, *SSL_read()* will only return, once the read operation has been finished or an error occurred, except when a renegotiation take place, in which case a SSL_ERROR_WANT_READ may occur. This behaviour can be controlled with the SSL_MODE_AUTO_RETRY flag of the *SSL_CTX_set_mode* (3) call.

If the underlying BIO is **non-blocking**, *SSL_read()* will also return when the underlying BIO could not satisfy the needs of *SSL_read()* to continue the operation. In this case a call to *SSL_get_error* (3) with the return value of *SSL_read()* will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. As at any time a re-negotiation is possible, a call to *SSL_read()* can also cause write operations! The calling process then must repeat the call after taking appropriate action to satisfy the needs of *SSL_read()*. The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but *select()* can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

WARNING

When an *SSL_read()* operation has to be repeated because of **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**, it must be repeated with the same arguments.

RETURN VALUES

The following return values can occur:

>0 The read operation was successful; the return value is the number of bytes actually read from the TLS/SSL connection.

0 The read operation was not successful. The reason may either be a clean shutdown due to a ‘‘close notify’’ alert sent by the peer (in which case the SSL_RECEIVED_SHUTDOWN flag in the ssl shutdown state is set (see *SSL_shutdown* (3), *SSL_set_shutdown* (3)). It is also possible, that the peer simply shut down the underlying transport and the shutdown is incomplete. Call *SSL_get_error()* with the return

value **ret** to find out, whether an error occurred or the connection was shut down cleanly (SSL_ERROR_ZERO_RETURN).

SSLv2 (deprecated) does not support a shutdown alert protocol, so it can only be detected, whether the underlying connection was closed. It cannot be checked, whether the closure was initiated by the peer or by something else.

<0 The read operation was not successful, because either an error occurred or action must be taken by the calling process. Call *SSL_get_error()* with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error (3), *SSL_write* (3), *SSL_CTX_set_mode* (3), *SSL_CTX_new* (3), *SSL_connect* (3), *SSL_accept* (3) *SSL_set_connect_state* (3), *SSL_shutdown* (3), *SSL_set_shutdown* (3), *ssl* (3), *openssl_bio* (3)

NAME

SSL_rstate_string, SSL_rstate_string_long − get textual description of state of an SSL object during read operation

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

const char *SSL_rstate_string(SSL *ssl);
const char *SSL_rstate_string_long(SSL *ssl);
```

DESCRIPTION

SSL_rstate_string() returns a 2 letter string indicating the current read state of the SSL object **ssl**.

SSL_rstate_string_long() returns a string indicating the current read state of the SSL object **ssl**.

NOTES

When performing a read operation, the SSL/TLS engine must parse the record, consisting of header and body. When working in a blocking environment, SSL_rstate_string[_long]() should always return ‘‘RD’’/‘‘read done’’.

This function should only seldom be needed in applications.

RETURN VALUES

SSL_rstate_string() and *SSL_rstate_string_long()* can return the following values:

‘‘RH’’/‘‘read header’’
: The header of the record is being evaluated.

‘‘RB’’/‘‘read body’’
: The body of the record is being evaluated.

‘‘RD’’/‘‘read done’’
: The record has been completely processed.

‘‘unknown’’/‘‘unknown’’
: The read state is unknown. This should never happen.

SEE ALSO

ssl (3)

NAME

SSL_session_reused − query whether a reused session was negotiated during handshake

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_session_reused(SSL *ssl);
```

DESCRIPTION

Query, whether a reused session was negotiated during the handshake.

NOTES

During the negotiation, a client can propose to reuse a session. The server then looks up the session in its cache. If both client and server agree on the session, it will be reused and a flag is being set that can be queried by the application.

RETURN VALUES

The following return values can occur:

0 A new session was negotiated.

1 A session was reused.

SEE ALSO

ssl (3), *SSL_set_session* (3), *SSL_CTX_set_session_cache_mode* (3)

NAME

SSL_set_bio − connect the SSL object with a BIO

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_set_bio(SSL *ssl, BIO *rbio, BIO *wbio);
```

DESCRIPTION

SSL_set_bio() connects the BIOs **rbio** and **wbio** for the read and write operations of the TLS/SSL (encrypted) side of **ssl**.

The SSL engine inherits the behaviour of **rbio** and **wbio**, respectively. If a BIO is non−blocking, the **ssl** will also have non-blocking behaviour.

If there was already a BIO connected to **ssl**, *BIO_free()* will be called (for both the reading and writing side, if different).

RETURN VALUES

SSL_set_bio() cannot fail.

SEE ALSO

SSL_get_rbio (3), *SSL_connect* (3), *SSL_accept* (3), *SSL_shutdown* (3), *ssl* (3), *openssl_bio* (3)

NAME

SSL_set_connect_state, SSL_get_accept_state − prepare SSL object to work in client or server mode

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_set_connect_state(SSL *ssl);

void SSL_set_accept_state(SSL *ssl);
```

DESCRIPTION

SSL_set_connect_state() sets **ssl** to work in client mode.

SSL_set_accept_state() sets **ssl** to work in server mode.

NOTES

When the SSL_CTX object was created with *SSL_CTX_new* (3), it was either assigned a dedicated client method, a dedicated server method, or a generic method, that can be used for both client and server connections. (The method might have been changed with *SSL_CTX_set_ssl_version* (3) or *SSL_set_ssl_method()*.)

When beginning a new handshake, the SSL engine must know whether it must call the connect (client) or accept (server) routines. Even though it may be clear from the method chosen, whether client or server mode was requested, the handshake routines must be explicitly set.

When using the *SSL_connect* (3) or *SSL_accept* (3) routines, the correct handshake routines are automatically set. When performing a transparent negotiation using *SSL_write* (3) or *SSL_read* (3), the handshake routines must be explicitly set in advance using either *SSL_set_connect_state()* or *SSL_set_accept_state()*.

RETURN VALUES

SSL_set_connect_state() and *SSL_set_accept_state()* do not return diagnostic information.

SEE ALSO

ssl (3), *SSL_new* (3), *SSL_CTX_new* (3), *SSL_connect* (3), *SSL_accept* (3), *SSL_write* (3), *SSL_read* (3), *SSL_do_handshake* (3), *SSL_CTX_set_ssl_version* (3)

NAME

SSL_set_fd − connect the SSL object with a file descriptor

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_set_fd(SSL *ssl, int fd);
int SSL_set_rfd(SSL *ssl, int fd);
int SSL_set_wfd(SSL *ssl, int fd);
```

DESCRIPTION

SSL_set_fd() sets the file descriptor **fd** as the input/output facility for the TLS/SSL (encrypted) side of **ssl**. **fd** will typically be the socket file descriptor of a network connection.

When performing the operation, a **socket BIO** is automatically created to interface between the **ssl** and **fd**. The BIO and hence the SSL engine inherit the behaviour of **fd**. If **fd** is non−blocking, the **ssl** will also have non-blocking behaviour.

If there was already a BIO connected to **ssl**, *BIO_free()* will be called (for both the reading and writing side, if different).

SSL_set_rfd() and *SSL_set_wfd()* perform the respective action, but only for the read channel or the write channel, which can be set independently.

RETURN VALUES

The following return values can occur:

0 The operation failed. Check the error stack to find out why.

1 The operation succeeded.

SEE ALSO

SSL_get_fd (3), *SSL_set_bio* (3), *SSL_connect* (3), *SSL_accept* (3), *SSL_shutdown* (3), *ssl* (3) , *openssl_bio* (3)

NAME

SSL_set_session − set a TLS/SSL session to be used during TLS/SSL connect

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_set_session(SSL *ssl, SSL_SESSION *session);
```

DESCRIPTION

SSL_set_session() sets **session** to be used when the TLS/SSL connection is to be established. *SSL_set_session()* is only useful for TLS/SSL clients. When the session is set, the reference count of **session** is incremented by 1. If the session is not reused, the reference count is decremented again during *SSL_connect()*. Whether the session was reused can be queried with the *SSL_session_reused* (3) call.

If there is already a session set inside **ssl** (because it was set with *SSL_set_session()* before or because the same **ssl** was already used for a connection), *SSL_SESSION_free()* will be called for that session.

NOTES

SSL_SESSION objects keep internal link information about the session cache list, when being inserted into one SSL_CTX object's session cache. One SSL_SESSION object, regardless of its reference count, must therefore only be used with one SSL_CTX object (and the SSL objects created from this SSL_CTX object).

RETURN VALUES

The following return values can occur:

0 The operation failed; check the error stack to find out the reason.

1 The operation succeeded.

SEE ALSO

ssl (3), *SSL_SESSION_free* (3), *SSL_get_session* (3), *SSL_session_reused* (3), *SSL_CTX_set_session_cache_mode* (3)

NAME

SSL_set_shutdown, SSL_get_shutdown − manipulate shutdown state of an SSL connection

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_set_shutdown(SSL *ssl, int mode);

int SSL_get_shutdown(const SSL *ssl);
```

DESCRIPTION

SSL_set_shutdown() sets the shutdown state of **ssl** to **mode**.

SSL_get_shutdown() returns the shutdown mode of **ssl**.

NOTES

The shutdown state of an ssl connection is a bitmask of:

0 No shutdown setting, yet.

SSL_SENT_SHUTDOWN
: A ‘‘close notify’’ shutdown alert was sent to the peer, the connection is being considered closed and the session is closed and correct.

SSL_RECEIVED_SHUTDOWN
: A shutdown alert was received form the peer, either a normal ‘‘close notify’’ or a fatal error.

SSL_SENT_SHUTDOWN and SSL_RECEIVED_SHUTDOWN can be set at the same time.

The shutdown state of the connection is used to determine the state of the ssl session. If the session is still open, when *SSL_clear* (3) or *SSL_free* (3) is called, it is considered bad and removed according to RFC2246. The actual condition for a correctly closed session is SSL_SENT_SHUTDOWN (according to the TLS RFC, it is acceptable to only send the ‘‘close notify’’ alert but to not wait for the peer’s answer, when the underlying connection is closed). *SSL_set_shutdown()* can be used to set this state without sending a close alert to the peer (see *SSL_shutdown* (3)).

If a ‘‘close notify’’ was received, SSL_RECEIVED_SHUTDOWN will be set, for setting SSL_SENT_SHUTDOWN the application must however still call *SSL_shutdown* (3) or *SSL_set_shutdown()* itself.

RETURN VALUES

SSL_set_shutdown() does not return diagnostic information.

SSL_get_shutdown() returns the current setting.

SEE ALSO

ssl (3), *SSL_shutdown* (3), *SSL_CTX_set_quiet_shutdown* (3), *SSL_clear* (3), *SSL_free* (3)

NAME

SSL_set_verify_result − override result of peer certificate verification

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_set_verify_result(SSL *ssl, long verify_result);
```

DESCRIPTION

SSL_set_verify_result() sets **verify_result** of the object **ssl** to be the result of the verification of the X509 certificate presented by the peer, if any.

NOTES

SSL_set_verify_result() overrides the verification result. It only changes the verification result of the **ssl** object. It does not become part of the established session, so if the session is to be reused later, the original value will reappear.

The valid codes for **verify_result** are documented in *openssl_verify* (1).

RETURN VALUES

SSL_set_verify_result() does not provide a return value.

SEE ALSO

ssl (3), *SSL_get_verify_result* (3), *SSL_get_peer_certificate* (3), *openssl_verify* (1)

NAME

SSL_shutdown − shut down a TLS/SSL connection

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_shutdown(SSL *ssl);
```

DESCRIPTION

SSL_shutdown() shuts down an active TLS/SSL connection. It sends the ''close notify'' shutdown alert to the peer.

NOTES

SSL_shutdown() tries to send the ''close notify'' shutdown alert to the peer. Whether the operation succeeds or not, the SSL_SENT_SHUTDOWN flag is set and a currently open session is considered closed and good and will be kept in the session cache for further reuse.

The shutdown procedure consists of 2 steps: the sending of the ''close notify'' shutdown alert and the reception of the peer's ''close notify'' shutdown alert. According to the TLS standard, it is acceptable for an application to only send its shutdown alert and then close the underlying connection without waiting for the peer's response (this way resources can be saved, as the process can already terminate or serve another connection). When the underlying connection shall be used for more communications, the complete shutdown procedure (bidirectional ''close notify'' alerts) must be performed, so that the peers stay synchronized.

SSL_shutdown() supports both uni− and bidirectional shutdown by its 2 step behaviour.

When the application is the first party to send the ''close notify'' alert, *SSL_shutdown()* will only send the alert and then set the SSL_SENT_SHUTDOWN flag (so that the session is considered good and will be kept in cache). *SSL_shutdown()* will then return with 0. If a unidirectional shutdown is enough (the underlying connection shall be closed anyway), this first call to *SSL_shutdown()* is sufficient. In order to complete the bidirectional shutdown handshake, *SSL_shutdown()* must be called again. The second call will make *SSL_shutdown()* wait for the peer's ''close notify'' shutdown alert. On success, the second call to *SSL_shutdown()* will return with 1.
If the peer already sent the ''close notify'' alert **and** it was already processed implicitly inside another function (*SSL_read*(3)), the SSL_RECEIVED_SHUTDOWN flag is set. *SSL_shutdown()* will send the ''close notify'' alert, set the SSL_SENT_SHUTDOWN flag and will immediately return with 1. Whether SSL_RECEIVED_SHUTDOWN is already set can be checked using the *SSL_get_shutdown()* (see also *SSL_set_shutdown*(3) call.

It is therefore recommended, to check the return value of *SSL_shutdown()* and call *SSL_shutdown()* again, if the bidirectional shutdown is not yet complete (return value of the first call is 0). As the shutdown is not specially handled in the SSLv2 protocol, *SSL_shutdown()* will succeed on the first call.

The behaviour of *SSL_shutdown()* additionally depends on the underlying BIO.

If the underlying BIO is **blocking**, *SSL_shutdown()* will only return once the handshake step has been finished or an error occurred.

If the underlying BIO is **non-blocking**, *SSL_shutdown()* will also return when the underlying BIO could not satisfy the needs of *SSL_shutdown()* to continue the handshake. In this case a call to *SSL_get_error()* with the return value of *SSL_shutdown()* will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process then must repeat the call after taking appropriate action to satisfy the needs of *SSL_shutdown()*. The action depends on the underlying BIO. When using a nonblocking socket, nothing is to be done, but *select()* can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

SSL_shutdown() can be modified to only set the connection to ''shutdown'' state but not actually send the ''close notify'' alert messages, see *SSL_CTX_set_quiet_shutdown*(3). When ''quiet shutdown'' is enabled,

SSL_shutdown() will always succeed and return 1.

RETURN VALUES

The following return values can occur:

1 The shutdown was successfully completed. The ‘‘close notify’’ alert was sent and the peer’s ‘‘close notify’’ alert was received.

0 The shutdown is not yet finished. Call *SSL_shutdown()* for a second time, if a bidirectional shutdown shall be performed. The output of *SSL_get_error* (3) may be misleading, as an erroneous SSL_ERROR_SYSCALL may be flagged even though no error occurred.

−1 The shutdown was not successful because a fatal error occurred either at the protocol level or a connection failure occurred. It can also occur if action is need to continue the operation for non-blocking BIOs. Call *SSL_get_error* (3) with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error (3), *SSL_connect* (3), *SSL_accept* (3), *SSL_set_shutdown* (3), *SSL_CTX_set_quiet_shutdown* (3), *SSL_clear* (3), *SSL_free* (3), *ssl* (3), *openssl_bio* (3)

NAME

SSL_state_string, SSL_state_string_long − get textual description of state of an SSL object

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

const char *SSL_state_string(const SSL *ssl);
const char *SSL_state_string_long(const SSL *ssl);
```

DESCRIPTION

SSL_state_string() returns a 6 letter string indicating the current state of the SSL object **ssl**.

SSL_state_string_long() returns a string indicating the current state of the SSL object **ssl**.

NOTES

During its use, an SSL objects passes several states. The state is internally maintained. Querying the state information is not very informative before or when a connection has been established. It however can be of significant interest during the handshake.

When using non-blocking sockets, the function call performing the handshake may return with SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE condition, so that SSL_state_string[_long]() may be called.

For both blocking or non-blocking sockets, the details state information can be used within the info_callback function set with the *SSL_set_info_callback()* call.

RETURN VALUES

Detailed description of possible states to be included later.

SEE ALSO

ssl (3), *SSL_CTX_set_info_callback* (3)

NAME

SSL_want, SSL_want_nothing, SSL_want_read, SSL_want_write, SSL_want_x509_lookup − obtain state information TLS/SSL I/O operation

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_want(const SSL *ssl);
int SSL_want_nothing(const SSL *ssl);
int SSL_want_read(const SSL *ssl);
int SSL_want_write(const SSL *ssl);
int SSL_want_x509_lookup(const SSL *ssl);
```

DESCRIPTION

SSL_want() returns state information for the SSL object **ssl**.

The other SSL_want_*() calls are shortcuts for the possible states returned by *SSL_want()*.

NOTES

SSL_want() examines the internal state information of the SSL object. Its return values are similar to that of *SSL_get_error* (3). Unlike *SSL_get_error* (3), which also evaluates the error queue, the results are obtained by examining an internal state flag only. The information must therefore only be used for normal operation under non-blocking I/O. Error conditions are not handled and must be treated using *SSL_get_error* (3).

The result returned by *SSL_want()* should always be consistent with the result of *SSL_get_error* (3).

RETURN VALUES

The following return values can currently occur for *SSL_want()*:

SSL_NOTHING
: There is no data to be written or to be read.

SSL_WRITING
: There are data in the SSL buffer that must be written to the underlying **BIO** layer in order to complete the actual SSL_*() operation. A call to *SSL_get_error* (3) should return SSL_ERROR_WANT_WRITE.

SSL_READING
: More data must be read from the underlying **BIO** layer in order to complete the actual SSL_*() operation. A call to *SSL_get_error* (3) should return SSL_ERROR_WANT_READ.

SSL_X509_LOOKUP
: The operation did not complete because an application callback set by *SSL_CTX_set_client_cert_cb()* has asked to be called again. A call to *SSL_get_error* (3) should return SSL_ERROR_WANT_X509_LOOKUP.

SSL_want_nothing(), *SSL_want_read()*, *SSL_want_write()*, *SSL_want_x509_lookup()* return 1, when the corresponding condition is true or 0 otherwise.

SEE ALSO

ssl (3), *openssl_err* (3), *SSL_get_error* (3)

NAME

SSL_write − write bytes to a TLS/SSL connection.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_write(SSL *ssl, const void *buf, int num);
```

DESCRIPTION

SSL_write() writes **num** bytes from the buffer **buf** into the specified **ssl** connection.

NOTES

If necessary, *SSL_write()* will negotiate a TLS/SSL session, if not already explicitly performed by *SSL_connect* (3) or *SSL_accept* (3). If the peer requests a re−negotiation, it will be performed transparently during the *SSL_write()* operation. The behaviour of *SSL_write()* depends on the underlying BIO.

For the transparent negotiation to succeed, the **ssl** must have been initialized to client or server mode. This is being done by calling *SSL_set_connect_state* (3) or *SSL_set_accept_state()* before the first call to an *SSL_read* (3) or *SSL_write()* function.

If the underlying BIO is **blocking**, *SSL_write()* will only return, once the write operation has been finished or an error occurred, except when a renegotiation take place, in which case a SSL_ERROR_WANT_READ may occur. This behaviour can be controlled with the SSL_MODE_AUTO_RETRY flag of the *SSL_CTX_set_mode* (3) call.

If the underlying BIO is **non-blocking**, *SSL_write()* will also return, when the underlying BIO could not satisfy the needs of *SSL_write()* to continue the operation. In this case a call to *SSL_get_error* (3) with the return value of *SSL_write()* will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. As at any time a re-negotiation is possible, a call to *SSL_write()* can also cause read operations! The calling process then must repeat the call after taking appropriate action to satisfy the needs of *SSL_write()*. The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but *select()* can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

SSL_write() will only return with success, when the complete contents of **buf** of length **num** has been written. This default behaviour can be changed with the SSL_MODE_ENABLE_PARTIAL_WRITE option of *SSL_CTX_set_mode* (3). When this flag is set, *SSL_write()* will also return with success, when a partial write has been successfully completed. In this case the *SSL_write()* operation is considered completed. The bytes are sent and a new *SSL_write()* operation with a new buffer (with the already sent bytes removed) must be started. A partial write is performed with the size of a message block, which is 16kB for SSLv3/TLSv1.

WARNING

When an *SSL_write()* operation has to be repeated because of **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**, it must be repeated with the same arguments.

When calling *SSL_write()* with num=0 bytes to be sent the behaviour is undefined.

RETURN VALUES

The following return values can occur:

>0 The write operation was successful, the return value is the number of bytes actually written to the TLS/SSL connection.

0 The write operation was not successful. Probably the underlying connection was closed. Call *SSL_get_error()* with the return value **ret** to find out, whether an error occurred or the connection was shut down cleanly (SSL_ERROR_ZERO_RETURN).

SSLv2 (deprecated) does not support a shutdown alert protocol, so it can only be detected, whether the underlying connection was closed. It cannot be checked, why the closure happened.

<0 The write operation was not successful, because either an error occurred or action must be taken by the calling process. Call *SSL_get_error()* with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error (3), *SSL_read* (3), *SSL_CTX_set_mode* (3), *SSL_CTX_new* (3), *SSL_connect* (3), *SSL_accept* (3) *SSL_set_connect_state* (3), *ssl* (3), *openssl_bio* (3)

NAME

X509_NAME_ENTRY_get_object, X509_NAME_ENTRY_get_data, X509_NAME_ENTRY_set_object, X509_NAME_ENTRY_set_data, X509_NAME_ENTRY_create_by_txt, X509_NAME_ENTRY_create_by_NID, X509_NAME_ENTRY_create_by_OBJ − X509_NAME_ENTRY utility functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>

ASN1_OBJECT * X509_NAME_ENTRY_get_object(X509_NAME_ENTRY *ne);
ASN1_STRING * X509_NAME_ENTRY_get_data(X509_NAME_ENTRY *ne);

int X509_NAME_ENTRY_set_object(X509_NAME_ENTRY *ne, ASN1_OBJECT *obj);
int X509_NAME_ENTRY_set_data(X509_NAME_ENTRY *ne, int type, const unsigned char *by

X509_NAME_ENTRY *X509_NAME_ENTRY_create_by_txt(X509_NAME_ENTRY **ne, const char *fi
X509_NAME_ENTRY *X509_NAME_ENTRY_create_by_NID(X509_NAME_ENTRY **ne, int nid, int t
X509_NAME_ENTRY *X509_NAME_ENTRY_create_by_OBJ(X509_NAME_ENTRY **ne, ASN1_OBJECT *o
```

DESCRIPTION

X509_NAME_ENTRY_get_object() retrieves the field name of **ne** in and **ASN1_OBJECT** structure.

X509_NAME_ENTRY_get_data() retrieves the field value of **ne** in and **ASN1_STRING** structure.

X509_NAME_ENTRY_set_object() sets the field name of **ne** to **obj**.

X509_NAME_ENTRY_set_data() sets the field value of **ne** to string type **type** and value determined by **bytes** and **len**.

X509_NAME_ENTRY_create_by_txt(), *X509_NAME_ENTRY_create_by_NID()* and *X509_NAME_ENTRY_create_by_OBJ()* create and return an **X509_NAME_ENTRY** structure.

NOTES

X509_NAME_ENTRY_get_object() and *X509_NAME_ENTRY_get_data()* can be used to examine an **X509_NAME_ENTRY** function as returned by *X509_NAME_get_entry()* for example.

X509_NAME_ENTRY_create_by_txt(), *X509_NAME_ENTRY_create_by_NID()*, and *X509_NAME_ENTRY_create_by_OBJ()* create and return an

X509_NAME_ENTRY_create_by_txt(), *X509_NAME_ENTRY_create_by_OBJ()*, *X509_NAME_ENTRY_create_by_NID()* and *X509_NAME_ENTRY_set_data()* are seldom used in practice because **X509_NAME_ENTRY** structures are almost always part of **X509_NAME** structures and the corresponding **X509_NAME** functions are typically used to create and add new entries in a single operation.

The arguments of these functions support similar options to the similarly named ones of the corresponding **X509_NAME** functions such as *X509_NAME_add_entry_by_txt()*. So for example **type** can be set to **MBSTRING_ASC** but in the case of *X509_set_data()* the field name must be set first so the relevant field information can be looked up internally.

RETURN VALUES

SEE ALSO

ERR_get_error (3), *d2i_X509_NAME* (3), *OBJ_nid2obj* (3),*OBJ_nid2obj* (3)

HISTORY

TBA

NAME

X509_NAME_add_entry_by_txt, X509_NAME_add_entry_by_OBJ, X509_NAME_add_entry_by_NID, X509_NAME_add_entry, X509_NAME_delete_entry − X509_NAME modification functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>

int X509_NAME_add_entry_by_txt(X509_NAME *name, const char *field, int type, const

int X509_NAME_add_entry_by_OBJ(X509_NAME *name, ASN1_OBJECT *obj, int type, unsigne

int X509_NAME_add_entry_by_NID(X509_NAME *name, int nid, int type, unsigned char *b

int X509_NAME_add_entry(X509_NAME *name,X509_NAME_ENTRY *ne, int loc, int set);

X509_NAME_ENTRY *X509_NAME_delete_entry(X509_NAME *name, int loc);
```

DESCRIPTION

X509_NAME_add_entry_by_txt(), *X509_NAME_add_entry_by_OBJ()* and *X509_NAME_add_entry_by_NID()* add a field whose name is defined by a string **field**, an object **obj** or a NID **nid** respectively. The field value to be added is in **bytes** of length **len**. If **len** is −1 then the field length is calculated internally using strlen(bytes).

The type of field is determined by **type** which can either be a definition of the type of **bytes** (such as **MBSTRING_ASC**) or a standard ASN1 type (such as **V_ASN1_IA5STRING**). The new entry is added to a position determined by **loc** and **set**.

X509_NAME_add_entry() adds a copy of **X509_NAME_ENTRY** structure **ne** to **name**. The new entry is added to a position determined by **loc** and **set**. Since a copy of **ne** is added **ne** must be freed up after the call.

X509_NAME_delete_entry() deletes an entry from **name** at position **loc**. The deleted entry is returned and must be freed up.

NOTES

The use of string types such as **MBSTRING_ASC** or **MBSTRING_UTF8** is strongly recommened for the **type** parameter. This allows the internal code to correctly determine the type of the field and to apply length checks according to the relevant standards. This is done using *ASN1_STRING_set_by_NID()*.

If instead an ASN1 type is used no checks are performed and the supplied data in **bytes** is used directly.

In *X509_NAME_add_entry_by_txt()* the **field** string represents the field name using OBJ_txt2obj(field, 0).

The **loc** and **set** parameters determine where a new entry should be added. For almost all applications **loc** can be set to −1 and **set** to 0. This adds a new entry to the end of **name** as a single valued RelativeDistinguishedName (RDN).

loc actually determines the index where the new entry is inserted: if it is −1 it is appended.

set determines how the new type is added. If it is zero a new RDN is created.

If **set** is −1 or 1 it is added to the previous or next RDN structure respectively. This will then be a multivalued RDN: since multivalues RDNs are very seldom used **set** is almost always set to zero.

EXAMPLES

Create an **X509_NAME** structure:

‘‘C=UK, O=Disorganized Organization, CN=Joe Bloggs’’

```
X509_NAME *nm;
nm = X509_NAME_new();
if (nm == NULL)
       /* Some error */
if (!X509_NAME_add_entry_by_txt(nm, MBSTRING_ASC,
                        "C", "UK", -1, -1, 0))
       /* Error */
if (!X509_NAME_add_entry_by_txt(nm, MBSTRING_ASC,
                        "O", "Disorganized Organization", -1, -1, 0))
       /* Error */
if (!X509_NAME_add_entry_by_txt(nm, MBSTRING_ASC,
                        "CN", "Joe Bloggs", -1, -1, 0))
       /* Error */
```

RETURN VALUES

X509_NAME_add_entry_by_txt(), *X509_NAME_add_entry_by_OBJ()*, *X509_NAME_add_entry_by_NID()* and *X509_NAME_add_entry()* return 1 for success of 0 if an error occurred.

X509_NAME_delete_entry() returns either the deleted **X509_NAME_ENTRY** structure of **NULL** if an error occurred.

BUGS

type can still be set to **V_ASN1_APP_CHOOSE** to use a different algorithm to determine field types. Since this form does not understand multicharacter types, performs no length checks and can result in invalid field types its use is strongly discouraged.

SEE ALSO

ERR_get_error (3), *d2i_X509_NAME* (3)

HISTORY

NAME

X509_NAME_get_index_by_NID, X509_NAME_get_index_by_OBJ, X509_NAME_get_entry, X509_NAME_entry_count, X509_NAME_get_text_by_NID, X509_NAME_get_text_by_OBJ − X509_NAME lookup and enumeration functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>

int X509_NAME_get_index_by_NID(X509_NAME *name,int nid,int lastpos);
int X509_NAME_get_index_by_OBJ(X509_NAME *name,ASN1_OBJECT *obj, int lastpos);

int X509_NAME_entry_count(X509_NAME *name);
X509_NAME_ENTRY *X509_NAME_get_entry(X509_NAME *name, int loc);

int X509_NAME_get_text_by_NID(X509_NAME *name, int nid, char *buf,int len);
int X509_NAME_get_text_by_OBJ(X509_NAME *name, ASN1_OBJECT *obj, char *buf,int len)
```

DESCRIPTION

These functions allow an **X509_NAME** structure to be examined. The **X509_NAME** structure is the same as the **Name** type defined in RFC2459 (and elsewhere) and used for example in certificate subject and issuer names.

X509_NAME_get_index_by_NID() and *X509_NAME_get_index_by_OBJ()* retrieve the next index matching **nid** or **obj** after **lastpos**. **lastpos** should initially be set to −1. If there are no more entries −1 is returned.

X509_NAME_entry_count() returns the total number of entries in **name**.

X509_NAME_get_entry() retrieves the **X509_NAME_ENTRY** from **name** corresponding to index **loc**. Acceptable values for **loc** run from 0 to (X509_NAME_entry_count(name) − 1). The value returned is an internal pointer which must not be freed.

X509_NAME_get_text_by_NID(), *X509_NAME_get_text_by_OBJ()* retrieve the ‘‘text’’ from the first entry in **name** which matches **nid** or **obj**, if no such entry exists −1 is returned. At most **len** bytes will be written and the text written to **buf** will be null terminated. The length of the output string written is returned excluding the terminating null. If **buf** is <NULL> then the amount of space needed in **buf** (excluding the final null) is returned.

NOTES

X509_NAME_get_text_by_NID() and *X509_NAME_get_text_by_OBJ()* are legacy functions which have various limitations which make them of minimal use in practice. They can only find the first matching entry and will copy the contents of the field verbatim: this can be highly confusing if the target is a muticharacter string type like a BMPString or a UTF8String.

For a more general solution *X509_NAME_get_index_by_NID()* or *X509_NAME_get_index_by_OBJ()* should be used followed by *X509_NAME_get_entry()* on any matching indices and then the various **X509_NAME_ENTRY** utility functions on the result.

EXAMPLES

Process all entries:

```
int i;
X509_NAME_ENTRY *e;

for (i = 0; i < X509_NAME_entry_count(nm); i++)
       {
       e = X509_NAME_get_entry(nm, i);
       /* Do something with e */
       }
```

Process all commonName entries:

```
int loc;
X509_NAME_ENTRY *e;

loc = -1;
for (;;)
        {
        lastpos = X509_NAME_get_index_by_NID(nm, NID_commonName, lastpos);
        if (lastpos == -1)
                break;
        e = X509_NAME_get_entry(nm, lastpos);
        /* Do something with e */
        }
```

RETURN VALUES

X509_NAME_get_index_by_NID() and *X509_NAME_get_index_by_OBJ()* return the index of the next matching entry or −1 if not found.

X509_NAME_entry_count() returns the total number of entries.

X509_NAME_get_entry() returns an **X509_NAME** pointer to the requested entry or **NULL** if the index is invalid.

SEE ALSO

ERR_get_error (3), *d2i_X509_NAME* (3)

HISTORY

TBA

NAME

X509_NAME_print_ex, X509_NAME_print_ex_fp, X509_NAME_print, X509_NAME_oneline − X509_NAME printing routines.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>

int X509_NAME_print_ex(BIO *out, X509_NAME *nm, int indent, unsigned long flags);
int X509_NAME_print_ex_fp(FILE *fp, X509_NAME *nm, int indent, unsigned long flags)
char * X509_NAME_oneline(X509_NAME *a,char *buf,int size);
int X509_NAME_print(BIO *bp, X509_NAME *name, int obase);
```

DESCRIPTION

X509_NAME_print_ex() prints a human readable version of **nm** to BIO **out**. Each line (for multiline formats) is indented by **indent** spaces. The output format can be extensively customised by use of the **flags** parameter.

X509_NAME_print_ex_fp() is identical to *X509_NAME_print_ex()* except the output is written to FILE pointer **fp**.

X509_NAME_oneline() prints an ASCII version of **a** to **buf**. At most **size** bytes will be written. If **buf** is **NULL** then a buffer is dynamically allocated and returned, otherwise **buf** is returned.

X509_NAME_print() prints out **name** to **bp** indenting each line by **obase** characters. Multiple lines are used if the output (including indent) exceeds 80 characters.

NOTES

The functions *X509_NAME_oneline()* and *X509_NAME_print()* are legacy functions which produce a non standard output form, they don't handle multi character fields and have various quirks and inconsistencies. Their use is strongly discouraged in new applications.

Although there are a large number of possible flags for most purposes **XN_FLAG_ONELINE**, **XN_FLAG_MULTILINE** or **XN_FLAG_RFC2253** will suffice. As noted on the *ASN1_STRING_print_ex* (3) manual page for UTF8 terminals the **ASN1_STRFLGS_ESC_MSB** should be unset: so for example **XN_FLAG_ONELINE & ~ASN1_STRFLGS_ESC_MSB** would be used.

The complete set of the flags supported by *X509_NAME_print_ex()* is listed below.

Several options can be ored together.

The options **XN_FLAG_SEP_COMMA_PLUS**, **XN_FLAG_SEP_CPLUS_SPC**, **XN_FLAG_SEP_SPLUS_SPC** and **XN_FLAG_SEP_MULTILINE** determine the field separators to use. Two distinct separators are used between distinct RelativeDistinguishedName components and separate values in the same RDN for a multi-valued RDN. Multi-valued RDNs are currently very rare so the second separator will hardly ever be used.

XN_FLAG_SEP_COMMA_PLUS uses comma and plus as separators. **XN_FLAG_SEP_CPLUS_SPC** uses comma and plus with spaces: this is more readable that plain comma and plus. **XN_FLAG_SEP_SPLUS_SPC** uses spaced semicolon and plus. **XN_FLAG_SEP_MULTILINE** uses spaced newline and plus respectively.

If **XN_FLAG_DN_REV** is set the whole DN is printed in reversed order.

The fields **XN_FLAG_FN_SN**, **XN_FLAG_FN_LN**, **XN_FLAG_FN_OID**, **XN_FLAG_FN_NONE** determine how a field name is displayed. It will use the short name (e.g. CN) the long name (e.g. commonName) always use OID numerical form (normally OIDs are only used if the field name is not recognised) and no field name respectively.

If **XN_FLAG_SPC_EQ** is set then spaces will be placed around the '=' character separating field names and values.

If **XN_FLAG_DUMP_UNKNOWN_FIELDS** is set then the encoding of unknown fields is printed instead of the values.

If **XN_FLAG_FN_ALIGN** is set then field names are padded to 20 characters: this is only of use for multi-line format.

Additionally all the options supported by *ASN1_STRING_print_ex()* can be used to control how each field value is displayed.

In addition a number options can be set for commonly used formats.

XN_FLAG_RFC2253 sets options which produce an output compatible with RFC2253 it is equivalent to:
ASN1_STRFLGS_RFC2253 | **XN_FLAG_SEP_COMMA_PLUS** | **XN_FLAG_DN_REV** | **XN_FLAG_FN_SN** | **XN_FLAG_DUMP_UNKNOWN_FIELDS**

XN_FLAG_ONELINE is a more readable one line format which is the same as:
ASN1_STRFLGS_RFC2253 | **ASN1_STRFLGS_ESC_QUOTE** | **XN_FLAG_SEP_CPLUS_SPC** | **XN_FLAG_SPC_EQ** | **XN_FLAG_FN_SN**

XN_FLAG_MULTILINE is a multiline format which is the same as:
ASN1_STRFLGS_ESC_CTRL | **ASN1_STRFLGS_ESC_MSB** | **XN_FLAG_SEP_MULTILINE** | **XN_FLAG_SPC_EQ** | **XN_FLAG_FN_LN** | **XN_FLAG_FN_ALIGN**

XN_FLAG_COMPAT uses a format identical to *X509_NAME_print()*: in fact it calls *X509_NAME_print()* internally.

SEE ALSO

ASN1_STRING_print_ex (3)

HISTORY

TBA

NAME

X509_new, X509_free − X509 certificate ASN1 allocation functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>

X509 *X509_new(void);
void X509_free(X509 *a);
```

DESCRIPTION

The X509 ASN1 allocation routines, allocate and free an X509 structure, which represents an X509 certificate.

X509_new() allocates and initializes a X509 structure.

X509_free() frees up the **X509** structure **a**.

RETURN VALUES

If the allocation fails, *X509_new()* returns **NULL** and sets an error code that can be obtained by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

X509_free() returns no value.

SEE ALSO

ERR_get_error (3), *d2i_X509* (3)

HISTORY

X509_new() and *X509_free()* are available in all versions of SSLeay and OpenSSL.

NAME

_DIAGASSERT — expression verification macro

SYNOPSIS

```
#include <assert.h>

_DIAGASSERT(expression);
```

DESCRIPTION

The **_DIAGASSERT**() macro tests the given *expression* and if it is false, one or more of the following may occur:

- a diagnostic message may be logged to the system logger with `syslog`(3). This is default behaviour.
- a diagnostic message may be printed to the `stderr` stream.
- the calling process will be terminated by calling `abort`(3).

This behaviour may be changed by setting the `LIBC_DIAGASSERT` environment variable (see below).

The diagnostic message consists of the text of the expression, the name of the source file, the line number and the enclosing function.

If *expression* is true, the **_DIAGASSERT**() macro does nothing.

The **_DIAGASSERT**() macro is not compiled in by default, and will only be compiled in with the `cc`(1) option **-D_DIAGNOSTIC**.

This macro is used in the various system libraries such as the Standard C Library (libc, −lc) to ensure that various library calls are invoked with valid arguments.

ENVIRONMENT

The `LIBC_DIAGASSERT` environment variable can be used to modify the default behaviour of logging the assertion to the system logger.

`LIBC_DIAGASSERT` may be set to one or more of the following characters:

a `abort`(3) once any assertion messages have been logged and/or printed.

A Opposite of “a”.

e Print the assertion message to the `stderr` stream.

E Opposite of “e”.

l Log the assertion message with `syslog`(3) to the facility `user.debug`.

L Opposite of “l”.

DIAGNOSTICS

The diagnostic message has the following format:

```
"assertion \"%s\" failed: file \"%s\", line %d, function \"%s\"\n",
           "expression", __FILE__, __LINE__, __func__
```

SEE ALSO

`cc`(1), `abort`(3), `assert`(3), `syslog`(3)

HISTORY

The **_DIAGASSERT** macro appeared in NetBSD 1.5.

NAME

__builtin_object_size — return the size of the given object

LIBRARY

size_t **__builtin_object_size**(*void *ptr*, *int type*)

DESCRIPTION

The **__builtin_object_size**() function is a gcc(1) built-in function that returns the size of the *ptr* object if known at compile time and the object does not have any side effects.

RETURN VALUES

If the size of the object is not known or it has side effects the **__builtin_object_size**() function returns:

`(size_t)-1` for *type* `0` and `1`.

`(size_t)0` for *type* `2` and `3`.

If the size of the object is known, then the **__builtin_object_size**() function returns the maximum size of all the objects that the compiler knows that they can be pointed to by *ptr* when *type* `& 2 == 0`, and the minimum size when *type* `& 2 != 0`.

SEE ALSO

gcc(1), ssp(3)

HISTORY

The **__builtin_object_size**() appeared in gcc 4.1.

NAME

_lwp_makecontext — create a new initial light-weight process execution context

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <lwp.h>

void
_lwp_makecontext(ucontext_t *context, void (*start_routine)(void *),
    void *arg, void *private, caddr_t stack_base, size_t stack_size);
```

DESCRIPTION

_lwp_makecontext() initializes the context structure pointed to by *context* in a manner suitable for using with _lwp_create(2). The LWP will begin execution at the function specified by *start_routine* which will be passed a single argument *arg*. The LWP private data pointer will be set to *private*. The stack region for the new LWP is specified by the *stack_base* and *stack_size* arguments.

The signal mask in the context structure is not initialized by **_lwp_makecontext**().

SEE ALSO

_lwp_create(2), _lwp_getprivate(2)

HISTORY

The **_lwp_create**() system call first appeared in NetBSD 2.0.

BUGS

The LWP private data pointer is not initialized by the current implementation of **_lwp_makecontext**().

NAME

a64l, **l64a**, **l64a_r** — convert between a long integer and a base-64 ASCII string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

long
a64l(const char *s);

char *
l64a(long int l);

int
l64a_r(long int l, char *buffer, int buflen);
```

DESCRIPTION

The **a64l**() and **l64a**() functions convert between a long integer and its base-64 ASCII string representation.

The characters used to represent ‘‘digits’’ are ‘.’ for 0, ‘/’ for 1, ‘0’ - ‘9’ for 2 - 11, ‘A’ - ‘Z’ for 12 - 37, and ‘a’ - ‘z’ for 38 - 63.

a64l() takes a pointer to a NUL-terminated base-64 ASCII string representation, *s*, and returns the corresponding long integer value.

l64a() takes a long integer value, *l*, and returns a pointer to the corresponding NUL-terminated base-64 ASCII string representation.

l64a_r() performs a conversion identical to that of **l64a**() and stores the resulting representation in the memory area pointed to by *buffer*, consuming at most *buflen* characters including the terminating NUL character.

RETURN VALUES

On successful completion, **a64l**() returns the long integer value corresponding to the input string. If the string pointed to by *s* is an empty string, **a64l**() returns a value of 0L.

l64a() returns a pointer to the base-64 ASCII string representation corresponding to the input value. If *l* is 0L, **l64a**() returns a pointer to an empty string.

On successful completion, **l64a_r**() returns 0; if *buffer* is of insufficient length, -1 is returned.

SEE ALSO

strtol(3)

STANDARDS

The **a64l**() and **l64a**() functions conform to X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”). The **l64a_r**() function conforms to System V Interface Definition, Fourth Edition (“SVID4”), Multithreading Extension.

BUGS

The **l64a**() function is not reentrant. The value returned by it points into a static buffer area; subsequent calls to **la64a**() may overwrite this buffer. In multi-threaded applications, **l64a_r**() should be used instead.

NAME

abort — cause abnormal program termination

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

void
abort(void);
```

DESCRIPTION

The **abort**() function causes abnormal program termination to occur, unless the signal `SIGABRT` is being caught and the signal handler does not return.

Calling the **abort**() function results in temporary files being removed. Any open streams are flushed and closed.

RETURN VALUES

The **abort** function never returns.

SEE ALSO

`sigaction`(2), `exit`(3)

STANDARDS

The **abort**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

abs — integer absolute value function

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
abs(int j);
```

DESCRIPTION

The **abs**() function computes the absolute value of the integer *j*.

RETURN VALUES

The **abs**() function returns the absolute value.

SEE ALSO

cabs(3), fabs(3), floor(3), hypot(3), labs(3), llabs(3), math(3)

STANDARDS

The **abs**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

BUGS

The absolute value of the most negative integer remains negative.

NAME

acos, **acosf** — arc cosine function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
acos(double x);

float
acosf(float x);
```

DESCRIPTION

The **acos**() and **acosf**() functions compute the principal value of the arc cosine of x in the range $[0, \pi]$.

RETURN VALUES

If |x|>1, **acos**(x) and **acosf**(x) set the global variable *errno* to EDOM.

SEE ALSO

asin(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **acos**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

acosh, **acoshf** — inverse hyperbolic cosine function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
acosh(double x);

float
acoshf(float x);
```

DESCRIPTION

The **acosh**() and **acoshf**() functions compute the inverse hyperbolic cosine of the real argument *x*.

RETURN VALUES

If x is less than one, **acosh**(*x*) and **acoshf**(*x*) return NaN and set the global variable *errno* to EDOM.

SEE ALSO

asinh(3), atanh(3), exp(3), math(3)

HISTORY

The **acosh**() function appeared in 4.3BSD.

NAME

aio_cancel — cancel an outstanding asynchronous I/O operation (REALTIME)

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <aio.h>

int
aio_cancel(int fildes, struct aiocb * aiocbp);
```

DESCRIPTION

The **aio_cancel**() system call cancels the outstanding asynchronous I/O request for the file descriptor specified in *fildes*. If *aiocbp* is specified, only that specific asynchronous I/O request is cancelled.

Normal asynchronous notification occurs for cancelled requests. Requests complete with an error result of ECANCELED.

RESTRICTIONS

The **aio_cancel**() system call does not cancel asynchronous I/O requests for raw disk devices. The **aio_cancel**() system call will always return AIO_NOTCANCELED for file descriptors associated with raw disk devices.

RETURN VALUES

The **aio_cancel**() system call returns −1 to indicate an error, or one of the following:

[AIO_CANCELED]
: All outstanding requests meeting the criteria specified were cancelled.

[AIO_NOTCANCELED]
: Some requests were not cancelled, status for the requests should be checked with aio_error(3).

[AIO_ALLDONE]
: All of the requests meeting the criteria have finished.

ERRORS

An error return from **aio_cancel**() indicates:

[EBADF] The *fildes* argument is an invalid file descriptor.

SEE ALSO

aio_error(3), aio_read(3), aio_return(3), aio_suspend(3), aio_write(3)

STANDARDS

The **aio_cancel**() system call is expected to conform to the IEEE Std 1003.1-2001 (“POSIX.1”) standard.

HISTORY

The **aio_cancel**() system call first appeared in NetBSD 5.0.

NAME

aio_error — retrieve error status of asynchronous I/O operation (REALTIME)

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <aio.h>

int
aio_error(const struct aiocb *aiocbp);
```

DESCRIPTION

The **aio_error**() system call returns the error status of the asynchronous I/O request associated with the structure pointed to by *aiocbp*.

RETURN VALUES

If the asynchronous I/O request has completed successfully, **aio_error**() returns 0. If the request has not yet completed, EINPROGRESS is returned. If the request has completed unsuccessfully the error status is returned as described in read(2), write(2), or fsync(2). On failure, **aio_error**() returns −1 and sets *errno* to indicate the error condition.

ERRORS

The **aio_error**() system call will fail if:

[EINVAL] The *aiocb* argument does not reference an outstanding asynchronous I/O request.

SEE ALSO

fsync(2), read(2), write(2), aio_cancel(3), aio_read(3), aio_return(3), aio_suspend(3), aio_write(3)

STANDARDS

The **aio_error**() system call is expected to conform to the IEEE Std 1003.1-2001 (“POSIX.1”) standard.

HISTORY

The **aio_error**() system call first appeared in NetBSD 5.0.

NAME

aio_fsync — asynchronous data synchronization of file (REALTIME)

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <aio.h>

int
aio_fsync(int op, struct aiocb *aiocbp);
```

DESCRIPTION

The **aio_fsync**() system call allows the calling process to force all modified data associated with the file descriptor *aiocbp->aio_fildes* to be flushed to the stable storage device. The call returns immediately after the synchronization request has been enqueued to the descriptor; the synchronization may or may not have completed at the time the call returns. If the request could not be enqueued, generally due to invalid arguments, the call returns without having enqueued the request.

The *op* argument could be set only to O_DSYNC or O_SYNC. If *op* is O_DSYNC, then **aio_fsync**() does the same as a **fdatasync**() call, if O_SYNC, then the same as **fsync**().

If _POSIX_PRIORITIZED_IO is defined, and the descriptor supports it, then the enqueued operation is submitted at a priority equal to that of the calling process minus *aiocbp->aio_reqprio*.

The *aiocbp* pointer may be subsequently used as an argument to **aio_return**() and **aio_error**() in order to determine return or error status for the enqueued operation while it is in progress.

RESTRICTIONS

The asynchronous I/O control buffer *aiocbp* should be zeroed before the **aio_fsync**() system call to avoid passing bogus context information to the kernel.

Modifications of the Asynchronous I/O Control Block structure after the request has been enqueued, but before the request has completed, are not allowed.

RETURN VALUES

The **aio_fsync**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **aio_fsync**() system call will fail if:

[EAGAIN] The request was not queued because of system resource limitations.

The following conditions may be synchronously detected when the **aio_fsync**() system call is made, or asynchronously, at any time thereafter. If they are detected at call time, **aio_fsync**() returns −1 and sets *errno* appropriately; otherwise the **aio_return**() system call must be called, and will return −1, and **aio_error**() must be called to determine the actual value that would have been returned in *errno*.

[EBADF] The *aiocbp->aio_fildes* is invalid for writing.

[EINVAL] This implementation does not support synchronized I/O for this file.

[EINVAL] The *op* argument is neither set to O_DSYNC nor O_SYNC.

SEE ALSO

`fcntl`(2), `fdatasync`(2), `fsync`(2), `aio_error`(3), `aio_read`(3), `aio_write`(3)

STANDARDS

The **`aio_fsync`**() system call is expected to conform to the IEEE Std 1003.1-2001 (“POSIX.1”) standard.

HISTORY

The **`aio_fsync`**() system call first appeared in NetBSD 5.0.

NAME

aio_read — asynchronous read from a file (REALTIME)

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <aio.h>

int
aio_read(struct aiocb *aiocbp);
```

DESCRIPTION

The **aio_read**() system call allows the calling process to read *aiocbp->aio_nbytes* from the descriptor *aiocbp->aio_fildes* beginning at the offset *aiocbp->aio_offset* into the buffer pointed to by *aiocbp->aio_buf*. The call returns immediately after the read request has been enqueued to the descriptor; the read may or may not have completed at the time the call returns.

If _POSIX_PRIORITIZED_IO is defined, and the descriptor supports it, then the enqueued operation is submitted at a priority equal to that of the calling process minus *aiocbp->aio_reqprio*.

The *aiocbp->aio_lio_opcode* argument is ignored by the **aio_read**() system call.

The *aiocbp* pointer may be subsequently used as an argument to **aio_return**() and **aio_error**() in order to determine return or error status for the enqueued operation while it is in progress.

If the request could not be enqueued (generally due to invalid arguments), then the call returns without having enqueued the request.

If the request is successfully enqueued, the value of *aiocbp->aio_offset* can be modified during the request as context, so this value must not be referenced after the request is enqueued.

RESTRICTIONS

The Asynchronous I/O Control Block structure pointed to by *aiocbp* and the buffer that the *aiocbp->aio_buf* member of that structure references must remain valid until the operation has completed. For this reason, use of auto (stack) variables for these objects is discouraged.

The asynchronous I/O control buffer *aiocbp* should be zeroed before the **aio_read**() call to avoid passing bogus context information to the kernel.

Modifications of the Asynchronous I/O Control Block structure or the buffer contents after the request has been enqueued, but before the request has completed, are not allowed.

If the file offset in *aiocbp->aio_offset* is past the offset maximum for *aiocbp->aio_fildes*, no I/O will occur.

RETURN VALUES

The **aio_read**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

DIAGNOSTICS

None.

ERRORS

The **aio_read**() system call will fail if:

[EAGAIN] The request was not queued because of system resource limitations.

The following conditions may be synchronously detected when the **aio_read**() system call is made, or asynchronously, at any time thereafter. If they are detected at call time, **aio_read**() returns −1 and sets *errno* appropriately; otherwise the **aio_return**() system call must be called, and will return −1, and **aio_error**() must be called to determine the actual value that would have been returned in *errno*.

[EBADF] The *aiocbp->aio_fildes* argument is invalid.

[EINVAL] The offset *aiocbp->aio_offset* is not valid, the priority specified by *aiocbp->aio_reqprio* is not a valid priority, or the number of bytes specified by *aiocbp->aio_nbytes* is not valid.

[EOVERFLOW] The file is a regular file, *aiocbp->aio_nbytes* is greater than zero, the starting offset in *aiocbp->aio_offset* is before the end of the file, but is at or beyond the *aiocbp->aio_fildes* offset maximum.

If the request is successfully enqueued, but subsequently cancelled or an error occurs, the value returned by the **aio_return**() system call is per the read(2) system call, and the value returned by the **aio_error**() system call is either one of the error returns from the read(2) system call, or one of:

[EBADF] The *aiocbp->aio_fildes* argument is invalid for reading.

[ECANCELED] The request was explicitly cancelled via a call to **aio_cancel**().

[EINVAL] The offset *aiocbp->aio_offset* would be invalid.

SEE ALSO

siginfo(2), aio_cancel(3), aio_error(3), aio_return(3), aio_suspend(3), aio_write(3)

STANDARDS

The **aio_read**() system call is expected to conform to the IEEE Std 1003.1-2001 (“POSIX.1”) standard.

HISTORY

The **aio_read**() system call first appeared in NetBSD 5.0.

NAME

aio_return — retrieve return status of asynchronous I/O operation (REALTIME)

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <aio.h>

int
aio_return(struct aiocb *aiocbp);
```

DESCRIPTION

The **aio_return**() system call returns the final status of the asynchronous I/O request associated with the structure pointed to by *aiocbp*.

The **aio_return**() system call should only be called once, to obtain the final status of an asynchronous I/O operation once it has completed (aio_error(3) returns something other than EINPROGRESS).

RETURN VALUES

If the asynchronous I/O request has completed, the status is returned as described in read(2), write(2), or fsync(2). Otherwise, **aio_return**() returns −1 and sets *errno* to indicate the error condition.

ERRORS

The **aio_return**() system call will fail if:

[EINVAL] The *aiocbp* argument does not reference a completed asynchronous I/O request.

SEE ALSO

fsync(2), read(2), write(2), aio_cancel(3), aio_error(3), aio_suspend(3), aio_write(3)

STANDARDS

The **aio_return**() system call is expected to conform to the IEEE Std 1003.1-2001 (“POSIX.1”) standard.

HISTORY

The **aio_return**() system call first appeared in NetBSD 5.0.

NAME

aio_suspend — suspend until asynchronous I/O operations or timeout complete (REALTIME)

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <aio.h>

int
aio_suspend(const struct aiocb * const list[], int nent,
    const struct timespec * timeout);
```

DESCRIPTION

The **aio_suspend**() system call suspends the calling process until at least one of the specified asynchronous I/O requests have completed, a signal is delivered, or the *timeout* has passed.

The *list* argument is an array of *nent* pointers to asynchronous I/O requests. Array members containing null pointers will be silently ignored.

If *timeout* is not a null pointer, it specifies a maximum interval to suspend. If *timeout* is a null pointer, the suspend blocks indefinitely. To effect a poll, the *timeout* should point to a zero-value timespec structure.

RETURN VALUES

If one or more of the specified asynchronous I/O requests have completed, **aio_suspend**() returns 0. Otherwise it returns −1 and sets *errno* to indicate the error, as enumerated below.

ERRORS

The **aio_suspend**() system call will fail if:

[EAGAIN] The *timeout* expired before any I/O requests completed.

[EINTR] The suspend was interrupted by a signal.

[EINVAL] The *list* argument contains more than AIO_LISTIO_MAX asynchronous I/O requests, or at least one of the requests is not valid.

SEE ALSO

aio_cancel(3), aio_error(3), aio_return(3), aio_write(3)

STANDARDS

The **aio_suspend**() system call is expected to conform to the IEEE Std 1003.1-2001 (“POSIX.1”) standard.

HISTORY

The **aio_suspend**() system call first appeared in NetBSD 5.0.

NAME

aio_write — asynchronous write to a file (REALTIME)

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <aio.h>

int
aio_write(struct aiocb *aiocbp);
```

DESCRIPTION

The **aio_write**() system call allows the calling process to write *aiocbp->aio_nbytes* from the buffer pointed to by *aiocbp->aio_buf* to the descriptor *aiocbp->aio_fildes*. The call returns immediately after the write request has been enqueued to the descriptor; the write may or may not have completed at the time the call returns. If the request could not be enqueued, generally due to invalid arguments, the call returns without having enqueued the request.

If O_APPEND is set for *aiocbp->aio_fildes*, **aio_write**() operations append to the file in the same order as the calls were made. If O_APPEND is not set for the file descriptor, the write operation will occur at the absolute position from the beginning of the file plus *aiocbp->aio_offset*.

If _POSIX_PRIORITIZED_IO is defined, and the descriptor supports it, then the enqueued operation is submitted at a priority equal to that of the calling process minus *aiocbp->aio_reqprio*.

The *aiocbp* pointer may be subsequently used as an argument to **aio_return**() and **aio_error**() in order to determine return or error status for the enqueued operation while it is in progress.

If the request is successfully enqueued, the value of *aiocbp->aio_offset* can be modified during the request as context, so this value must not be referenced after the request is enqueued.

RESTRICTIONS

The Asynchronous I/O Control Block structure pointed to by *aiocbp* and the buffer that the *aiocbp->aio_buf* member of that structure references must remain valid until the operation has completed. For this reason, use of auto (stack) variables for these objects is discouraged.

The asynchronous I/O control buffer *aiocbp* should be zeroed before the **aio_write**() system call to avoid passing bogus context information to the kernel.

Modifications of the Asynchronous I/O Control Block structure or the buffer contents after the request has been enqueued, but before the request has completed, are not allowed.

If the file offset in *aiocbp->aio_offset* is past the offset maximum for *aiocbp->aio_fildes*, no I/O will occur.

RETURN VALUES

The **aio_write**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **aio_write**() system call will fail if:

[EAGAIN] The request was not queued because of system resource limitations.

The following conditions may be synchronously detected when the **aio_write**() system call is made, or asynchronously, at any time thereafter. If they are detected at call time, **aio_write**() returns −1 and sets *errno* appropriately; otherwise the **aio_return**() system call must be called, and will return −1, and **aio_error**() must be called to determine the actual value that would have been returned in *errno*.

[EBADF] The *aiocbp->aio_fildes* argument is invalid, or is not opened for writing.

[EINVAL] The offset *aiocbp->aio_offset* is not valid, the priority specified by *aiocbp->aio_reqprio* is not a valid priority, or the number of bytes specified by *aiocbp->aio_nbytes* is not valid.

If the request is successfully enqueued, but subsequently canceled or an error occurs, the value returned by the **aio_return**() system call is per the write(2) system call, and the value returned by the **aio_error**() system call is either one of the error returns from the write(2) system call, or one of:

[EBADF] The *aiocbp->aio_fildes* argument is invalid for writing.

[ECANCELED] The request was explicitly canceled via a call to **aio_cancel**().

[EINVAL] The offset *aiocbp->aio_offset* would be invalid.

SEE ALSO

siginfo(2), aio_cancel(3), aio_error(3), aio_return(3), aio_suspend(3)

STANDARDS

The **aio_write**() system call is expected to conform to the IEEE Std 1003.1-2001 (“POSIX.1”) standard.

HISTORY

The **aio_write**() system call first appeared in NetBSD 5.0.

NAME

alarm — set signal timer alarm

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

unsigned int
alarm(unsigned int seconds);
```

DESCRIPTION

This interface is made obsolete by `setitimer`**(2).**

The **alarm**() function sets a timer to deliver the signal `SIGALRM` to the calling process *seconds* after the call to **alarm**(). If an alarm has already been set with **alarm**() but has not been delivered, another call to **alarm**() will supersede the prior call. The request **alarm**(*0*) voids the current alarm and the signal SIGALRM will not be delivered. The maximum number of *seconds* allowed is 2147483647.

The return value of **alarm**() is the amount of time left on the timer from a previous call to **alarm**(). If no alarm is currently set, the return value is 0. If there is an error setting the timer, **alarm**() returns ((unsigned int) -1).

SEE ALSO

`setitimer`(2), `sigaction`(2), `sigsuspend`(2), `signal`(3), `sigvec`(3), `sleep`(3), `ualarm`(3), `usleep`(3)

STANDARDS

The **alarm**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

An **alarm**() function appeared in Version 7 AT&T UNIX.

NAME

alloca — memory allocator

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

void *
alloca(size_t size);
```

DESCRIPTION

The **alloca**() function allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed on return.

RETURN VALUES

The **alloca**() function returns a pointer to the beginning of the allocated space. If the allocation failed, a NULL pointer is returned.

SEE ALSO

brk(2), calloc(3), getpagesize(3), malloc(3), realloc(3)

BUGS

The **alloca**() function is machine dependent; its use is discouraged.

The **alloca**() function is slightly unsafe because it cannot ensure that the pointer returned points to a valid and usable block of memory. The allocation made may exceed the bounds of the stack, or even go further into other objects in memory, and **alloca**() cannot determine such an error. Avoid **alloca**() with large unbounded allocations.

NAME

arc4random, **arc4random_stir**, **arc4random_addrandom** — arc4 random number generator

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

uint32_t
arc4random(void);

void
arc4random_stir(void);

void
arc4random_addrandom(u_char *dat, int datlen);
```

DESCRIPTION

The **arc4random**() function provides a high quality 32-bit pseudo-random number very quickly. **arc4random**() seeds itself on a regular basis from the kernel strong random number subsystem described in rnd(4). On each call, an ARC4 generator is used to generate a new result. The **arc4random**() function uses the ARC4 cipher key stream generator, which uses 8∗8 8 bit S-Boxes. The S-Boxes can be in about (2∗∗1700) states.

arc4random() fits into a middle ground not covered by other subsystems such as the strong, slow, and resource expensive random devices described in rnd(4) versus the fast but poor quality interfaces described in rand(3), random(3), and drand48(3).

The **arc4random_stir**() function reads data from /dev/urandom and uses it to permute the S-Boxes via **arc4random_addrandom**().

There is no need to call **arc4random_stir**() before using **arc4random**(), since **arc4random**() automatically initializes itself.

SEE ALSO

rand(3), rand48(3), random(3)

HISTORY

An algorithm called RC4 was designed by RSA Data Security, Inc. It was considered a trade secret, but not trademarked. Because it was a trade secret, it obviously could not be patented. A clone of this was posted anonymously to USENET and confirmed to be equivalent by several sources who had access to the original cipher. Because of the trade secret situation, RSA Data Security, Inc. can do nothing about the release of the ARC4 algorithm. Since RC4 used to be a trade secret, the cipher is now referred to as ARC4.

These functions first appeared in OpenBSD 2.1.

NAME

archive_entry_acl_add_entry, **archive_entry_acl_add_entry_w**, **archive_entry_acl_clear**, **archive_entry_acl_count**, **archive_entry_acl_next**, **archive_entry_acl_next_w**, **archive_entry_acl_reset**, **archive_entry_acl_text_w**, **archive_entry_atime**, **archive_entry_atime_nsec**, **archive_entry_clear**, **archive_entry_clone**, **archive_entry_copy_fflags_text**, **archive_entry_copy_fflags_text_w**, **archive_entry_copy_gname**, **archive_entry_copy_gname_w**, **archive_entry_copy_hardlink**, **archive_entry_copy_hardlink_w**, **archive_entry_copy_link**, **archive_entry_copy_link_w**, **archive_entry_copy_pathname_w**, **archive_entry_copy_sourcepath**, **archive_entry_copy_stat**, **archive_entry_copy_symlink**, **archive_entry_copy_symlink_w**, **archive_entry_copy_uname**, **archive_entry_copy_uname_w**, **archive_entry_dev**, **archive_entry_devmajor**, **archive_entry_devminor**, **archive_entry_filetype**, **archive_entry_fflags**, **archive_entry_fflags_text**, **archive_entry_free**, **archive_entry_gid**, **archive_entry_gname**, **archive_entry_hardlink**, **archive_entry_ino**, **archive_entry_mode**, **archive_entry_mtime**, **archive_entry_mtime_nsec**, **archive_entry_nlink**, **archive_entry_new**, **archive_entry_pathname**, **archive_entry_pathname_w**, **archive_entry_rdev**, **archive_entry_rdevmajor**, **archive_entry_rdevminor**, **archive_entry_set_atime**, **archive_entry_set_ctime**, **archive_entry_set_dev**, **archive_entry_set_devmajor**, **archive_entry_set_devminor**, **archive_entry_set_filetype**, **archive_entry_set_fflags**, **archive_entry_set_gid**, **archive_entry_set_gname**, **archive_entry_set_hardlink**, **archive_entry_set_link**, **archive_entry_set_mode**, **archive_entry_set_mtime**, **archive_entry_set_pathname**, **archive_entry_set_rdevmajor**, **archive_entry_set_rdevminor**, **archive_entry_set_size**, **archive_entry_set_symlink**, **archive_entry_set_uid**, **archive_entry_set_uname**, **archive_entry_size**, **archive_entry_sourcepath**, **archive_entry_stat**, **archive_entry_symlink**, **archive_entry_uid**, **archive_entry_uname** — functions for manipulating archive entry descriptions

SYNOPSIS

```
#include <archive_entry.h>

void
archive_entry_acl_add_entry(struct archive_entry *,int type,int permset,
    int tag,int qual,const char *name);

void
archive_entry_acl_add_entry_w(struct archive_entry *,int type,
    int permset,int tag,int qual,const wchar_t *name);

void
archive_entry_acl_clear(struct archive_entry *);

int
archive_entry_acl_count(struct archive_entry *, int type);

int
archive_entry_acl_next(struct archive_entry *,int want_type,int *type,
    int *permset,int *tag,int *qual,const char **name);

int
archive_entry_acl_next_w(struct archive_entry *,int want_type,int *type,
    int *permset,int *tag,int *qual,const wchar_t **name);
```

int
archive_entry_acl_reset(*struct archive_entry **, *int want_type*);

*const wchar_t **
archive_entry_acl_text_w(*struct archive_entry **, *int flags*);

time_t
archive_entry_atime(*struct archive_entry **);

long
archive_entry_atime_nsec(*struct archive_entry **);

*struct archive_entry **
archive_entry_clear(*struct archive_entry **);

*struct archive_entry **
archive_entry_clone(*struct archive_entry **);

*const char * **
archive_entry_copy_fflags_text_w(*struct archive_entry **, *const char **);

*const wchar_t **
archive_entry_copy_fflags_text_w(*struct archive_entry **, *const wchar_t **);

void
archive_entry_copy_gname(*struct archive_entry **, *const char **);

void
archive_entry_copy_gname_w(*struct archive_entry **, *const wchar_t **);

void
archive_entry_copy_hardlink(*struct archive_entry **, *const char **);

void
archive_entry_copy_hardlink_w(*struct archive_entry **, *const wchar_t **);

void
archive_entry_copy_sourcepath(*struct archive_entry **, *const char **);

void
archive_entry_copy_pathname_w(*struct archive_entry **, *const wchar_t **);

void
archive_entry_copy_stat(*struct archive_entry **, *const struct stat **);

void
archive_entry_copy_symlink(*struct archive_entry **, *const char **);

void
archive_entry_copy_symlink_w(*struct archive_entry **, *const wchar_t **);

void
archive_entry_copy_uname(*struct archive_entry **, *const char **);

void
archive_entry_copy_uname_w(*struct archive_entry **, *const wchar_t **);

dev_t
archive_entry_dev(*struct archive_entry **);

dev_t
archive_entry_devmajor(*struct archive_entry **);

dev_t
archive_entry_devminor(*struct archive_entry **);

mode_t
archive_entry_filetype(*struct archive_entry **);

void
archive_entry_fflags(*struct archive_entry **, *unsigned long *set*, *unsigned long *clear*);

*const char **
archive_entry_fflags_text(*struct archive_entry **);

void
archive_entry_free(*struct archive_entry **);

*const char **
archive_entry_gname(*struct archive_entry **);

*const char **
archive_entry_hardlink(*struct archive_entry **);

ino_t
archive_entry_ino(*struct archive_entry **);

mode_t
archive_entry_mode(*struct archive_entry **);

time_t
archive_entry_mtime(*struct archive_entry **);

long
archive_entry_mtime_nsec(*struct archive_entry **);

unsigned int
archive_entry_nlink(*struct archive_entry **);

*struct archive_entry **
archive_entry_new(*void*);

*const char **
archive_entry_pathname(*struct archive_entry **);

*const wchar_t **
archive_entry_pathname_w(*struct archive_entry **);

dev_t
archive_entry_rdev(*struct archive_entry **);

dev_t
archive_entry_rdevmajor(*struct archive_entry **);

dev_t
archive_entry_rdevminor(*struct archive_entry **);

void
archive_entry_set_dev(*struct archive_entry **, *dev_t*);

void
archive_entry_set_devmajor(*struct archive_entry **, *dev_t*);

void
archive_entry_set_devminor(*struct archive_entry **, *dev_t*);

void
archive_entry_set_filetype(*struct archive_entry **, *unsigned int*);

void
archive_entry_set_fflags(*struct archive_entry **,*unsigned long set*,
unsigned long clear);

void
archive_entry_set_gid(*struct archive_entry **, *gid_t*);

void
archive_entry_set_gname(*struct archive_entry **, *const char **);

void
archive_entry_set_hardlink(*struct archive_entry **, *const char **);

void
archive_entry_set_ino(*struct archive_entry **, *unsigned long*);

void
archive_entry_set_link(*struct archive_entry **, *const char **);

void
archive_entry_set_mode(*struct archive_entry **, *mode_t*);

void
archive_entry_set_mtime(*struct archive_entry **, *time_t*, *long nanos*);

void
archive_entry_set_nlink(*struct archive_entry **, *unsigned int*);

void
archive_entry_set_pathname(*struct archive_entry **, *const char **);

void
archive_entry_set_rdev(*struct archive_entry **, *dev_t*);

void
archive_entry_set_rdevmajor(*struct archive_entry **, *dev_t*);

void
archive_entry_set_rdevminor(*struct archive_entry **, *dev_t*);

void
archive_entry_set_size(*struct archive_entry **, *int64_t*);

void
archive_entry_set_symlink(*struct archive_entry **, *const char **);

void
archive_entry_set_uid(*struct archive_entry **, *uid_t*);

void
archive_entry_set_uname(*struct archive_entry **, *const char **);

int64_t
archive_entry_size(*struct archive_entry* *);

const char *
archive_entry_sourcepath(*struct archive_entry* *);

const struct stat *
archive_entry_stat(*struct archive_entry* *);

const char *
archive_entry_symlink(*struct archive_entry* *);

const char *
archive_entry_uname(*struct archive_entry* *);

DESCRIPTION

These functions create and manipulate data objects that represent entries within an archive. You can think of a struct archive_entry as a heavy-duty version of struct stat: it includes everything from struct stat plus associated pathname, textual group and user names, etc. These objects are used by `libarchive`(3) to represent the metadata associated with a particular entry in an archive.

Create and Destroy

There are functions to allocate, destroy, clear, and copy *archive_entry* objects:

archive_entry_clear()
: Erases the object, resetting all internal fields to the same state as a newly-created object. This is provided to allow you to quickly recycle objects without thrashing the heap.

archive_entry_clone()
: A deep copy operation; all text fields are duplicated.

archive_entry_free()
: Releases the struct archive_entry object.

archive_entry_new()
: Allocate and return a blank struct archive_entry object.

Set and Get Functions

Most of the functions here set or read entries in an object. Such functions have one of the following forms:

archive_entry_set_XXXX()
: Stores the provided data in the object. In particular, for strings, the pointer is stored, not the referenced string.

archive_entry_copy_XXXX()
: As above, except that the referenced data is copied into the object.

archive_entry_XXXX()
: Returns the specified data. In the case of strings, a const-qualified pointer to the string is returned.

String data can be set or accessed as wide character strings or normal *char* strings. The functions that use wide character strings are suffixed with **_w**. Note that these are different representations of the same data: For example, if you store a narrow string and read the corresponding wide string, the object will transparently convert formats using the current locale. Similarly, if you store a wide string and then store a narrow string for the same data, the previously-set wide string will be discarded in favor of the new data.

There are a few set/get functions that merit additional description:

archive_entry_set_link()
: This function sets the symlink field if it is already set. Otherwise, it sets the hardlink field.

File Flags

File flags are transparently converted between a bitmap representation and a textual format. For example, if you set the bitmap and ask for text, the library will build a canonical text format. However, if you set a text format and request a text format, you will get back the same text, even if it is ill-formed. If you need to canonicalize a textual flags string, you should first set the text form, then request the bitmap form, then use that to set the bitmap form. Setting the bitmap format will clear the internal text representation and force it to be reconstructed when you next request the text form.

The bitmap format consists of two integers, one containing bits that should be set, the other specifying bits that should be cleared. Bits not mentioned in either bitmap will be ignored. Usually, the bitmap of bits to be cleared will be set to zero. In unusual circumstances, you can force a fully-specified set of file flags by setting the bitmap of flags to clear to the complement of the bitmap of flags to set. (This differs from `fflagstostr`(3), which only includes names for set bits.) Converting a bitmap to a textual string is a platform-specific operation; bits that are not meaningful on the current platform will be ignored.

The canonical text format is a comma-separated list of flag names. The **`archive_entry_copy_fflags_text`**() and **`archive_entry_copy_fflags_text_w`**() functions parse the provided text and sets the internal bitmap values. This is a platform-specific operation; names that are not meaningful on the current platform will be ignored. The function returns a pointer to the start of the first name that was not recognized, or NULL if every name was recognized. Note that every name--including names that follow an unrecognized name--will be evaluated, and the bitmaps will be set to reflect every name that is recognized. (In particular, this differs from `strtofflags`(3), which stops parsing at the first unrecognized name.)

ACL Handling

XXX This needs serious help. XXX

An “Access Control List” (ACL) is a list of permissions that grant access to particular users or groups beyond what would normally be provided by standard POSIX mode bits. The ACL handling here addresses some deficiencies in the POSIX.1e draft 17 ACL specification. In particular, POSIX.1e draft 17 specifies several different formats, but none of those formats include both textual user/group names and numeric UIDs/GIDs.

XXX explain ACL stuff XXX

SEE ALSO

`archive`(3)

HISTORY

The **`libarchive`** library first appeared in FreeBSD 5.3.

AUTHORS

The **`libarchive`** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

NAME

archive_read_new, **archive_read_support_compression_all**, **archive_read_support_compression_bzip2**, **archive_read_support_compression_compress**, **archive_read_support_compression_gzip**, **archive_read_support_compression_none**, **archive_read_support_compression_program**, **archive_read_support_format_all**, **archive_read_support_format_cpio**, **archive_read_support_format_empty**, **archive_read_support_format_iso9660**, **archive_read_support_format_tar**, **archive_read_support_format_zip**, **archive_read_open**, **archive_read_open2**, **archive_read_open_fd**, **archive_read_open_FILE**, **archive_read_open_filename**, **archive_read_open_memory**, **archive_read_next_header**, **archive_read_data**, **archive_read_data_block**, **archive_read_data_skip**, **archive_read_data_into_buffer**, **archive_read_data_into_fd**, **archive_read_extract**, **archive_read_extract2**, **archive_read_extract_set_progress_callback**, **archive_read_close**, **archive_read_finish** — functions for reading streaming archives

SYNOPSIS

```
#include <archive.h>

struct archive *
archive_read_new(void);

int
archive_read_support_compression_all(struct archive *);

int
archive_read_support_compression_bzip2(struct archive *);

int
archive_read_support_compression_compress(struct archive *);

int
archive_read_support_compression_gzip(struct archive *);

int
archive_read_support_compression_none(struct archive *);

int
archive_read_support_compression_program(struct archive *,
    const char *cmd);

int
archive_read_support_format_all(struct archive *);

int
archive_read_support_format_cpio(struct archive *);

int
archive_read_support_format_empty(struct archive *);

int
archive_read_support_format_iso9660(struct archive *);

int
archive_read_support_format_tar(struct archive *);
```

int
archive_read_support_format_zip(*struct archive* *);

int
archive_read_open(*struct archive* *,*void* *client_data,
archive_open_callback *,*archive_read_callback* *,
archive_close_callback *);

int
archive_read_open2(*struct archive* *,*void* *client_data,
archive_open_callback *,*archive_read_callback* *,
archive_skip_callback *,*archive_close_callback* *);

int
archive_read_open_FILE(*struct archive* *, *FILE* *file);

int
archive_read_open_fd(*struct archive* *, *int fd*, *size_t block_size*);

int
archive_read_open_filename(*struct archive* *,*const char* *filename,
size_t block_size);

int
archive_read_open_memory(*struct archive* *, *void* *buff, *size_t size*);

int
archive_read_next_header(*struct archive* *, *struct archive_entry* **);

ssize_t
archive_read_data(*struct archive* *, *void* *buff, *size_t len*);

int
archive_read_data_block(*struct archive* *,*const void* **buff,*size_t* *len,
off_t *offset*);

int
archive_read_data_skip(*struct archive* *);

int
archive_read_data_into_buffer(*struct archive* *, *void* *, *ssize_t len*);

int
archive_read_data_into_fd(*struct archive* *, *int fd*);

int
archive_read_extract(*struct archive* *,*struct archive_entry* *,*int flags*);

int
archive_read_extract2(*struct archive* *src,*struct archive_entry* *,
struct archive *dest*);

void
archive_read_extract_set_progress_callback(*struct archive* *,
*void (*func)(void *)*,*void* *user_data*);

int
archive_read_close(*struct archive* *);

int
archive_read_finish(*struct archive* *);

DESCRIPTION

These functions provide a complete API for reading streaming archives. The general process is to first create the struct archive object, set options, initialize the reader, iterate over the archive headers and associated data, then close the archive and release all resources. The following summary describes the functions in approximately the order they would be used:

archive_read_new()
: Allocates and initializes a struct archive object suitable for reading from an archive.

archive_read_support_compression_all(),
archive_read_support_compression_bzip2(),
archive_read_support_compression_compress(),
archive_read_support_compression_gzip(),
archive_read_support_compression_none()
: Enables auto-detection code and decompression support for the specified compression. Note that "none" is always enabled by default. For convenience, **archive_read_support_compression_all**() enables all available decompression code.

archive_read_support_compression_program()
: Data is fed through the specified external program before being dearchived. Note that this disables automatic detection of the compression format, so it makes no sense to specify this in conjunction with any other decompression option.

archive_read_support_format_all(), **archive_read_support_format_cpio**(),
archive_read_support_format_empty(),
archive_read_support_format_iso9660(),
archive_read_support_format_tar(), **archive_read_support_format_zip**()
: Enables support---including auto-detection code---for the specified archive format. For example, **archive_read_support_format_tar**() enables support for a variety of standard tar formats, old-style tar, ustar, pax interchange format, and many common variants. For convenience, **archive_read_support_format_all**() enables support for all available formats. Only empty archives are supported by default.

archive_read_open()
: The same as **archive_read_open2**(), except that the skip callback is assumed to be NULL.

archive_read_open2()
: Freeze the settings, open the archive, and prepare for reading entries. This is the most generic version of this call, which accepts four callback functions. Most clients will want to use **archive_read_open_filename**(), **archive_read_open_FILE**(), **archive_read_open_fd**(), or **archive_read_open_memory**() instead. The library invokes the client-provided functions to obtain raw bytes from the archive.

archive_read_open_FILE()
: Like **archive_read_open**(), except that it accepts a *FILE* * pointer. This function should not be used with tape drives or other devices that require strict I/O blocking.

archive_read_open_fd()
: Like **archive_read_open**(), except that it accepts a file descriptor and block size rather than a set of function pointers. Note that the file descriptor will not be automatically closed at end-of-archive. This function is safe for use with tape drives or other blocked devices.

archive_read_open_file()
: This is a deprecated synonym for **archive_read_open_filename**().

archive_read_open_filename()
: Like **archive_read_open**(), except that it accepts a simple filename and a block size. A NULL filename represents standard input. This function is safe for use with tape drives or other blocked devices.

archive_read_open_memory()
Like **archive_read_open**(), except that it accepts a pointer and size of a block of memory containing the archive data.

archive_read_next_header()
Read the header for the next entry and return a pointer to a struct archive_entry.

archive_read_data()
Read data associated with the header just read. Internally, this is a convenience function that calls **archive_read_data_block**() and fills any gaps with nulls so that callers see a single continuous stream of data.

archive_read_data_block()
Return the next available block of data for this entry. Unlike **archive_read_data**(), the **archive_read_data_block**() function avoids copying data and allows you to correctly handle sparse files, as supported by some archive formats. The library guarantees that offsets will increase and that blocks will not overlap. Note that the blocks returned from this function can be much larger than the block size read from disk, due to compression and internal buffer optimizations.

archive_read_data_skip()
A convenience function that repeatedly calls **archive_read_data_block**() to skip all of the data for this archive entry.

archive_read_data_into_buffer()
This function is deprecated and will be removed. Use **archive_read_data**() instead.

archive_read_data_into_fd()
A convenience function that repeatedly calls **archive_read_data_block**() to copy the entire entry to the provided file descriptor.

archive_read_extract(), **archive_read_extract_set_skip_file**()
A convenience function that wraps the corresponding archive_write_disk(3) interfaces. The first call to **archive_read_extract**() creates a restore object using archive_write_disk_new(3) and archive_write_disk_set_standard_lookup(3), then transparently invokes archive_write_disk_set_options(3), archive_write_header(3), archive_write_data(3), and archive_write_finish_entry(3) to create the entry on disk and copy data into it. The *flags* argument is passed unmodified to archive_write_disk_set_options(3).

archive_read_extract2()
This is another version of **archive_read_extract**() that allows you to provide your own restore object. In particular, this allows you to override the standard lookup functions using archive_write_disk_set_group_lookup(3), and archive_write_disk_set_user_lookup(3). Note that **archive_read_extract2**() does not accept a *flags* argument; you should use **archive_write_disk_set_options**() to set the restore options yourself.

archive_read_extract_set_progress_callback()
Sets a pointer to a user-defined callback that can be used for updating progress displays during extraction. The progress function will be invoked during the extraction of large regular files. The progress function will be invoked with the pointer provided to this call. Generally, the data pointed to should include a reference to the archive object and the archive_entry object so that various statistics can be retrieved for the progress display.

archive_read_close()
Complete the archive and invoke the close callback.

archive_read_finish()
Invokes **archive_read_close**() if it was not invoked manually, then release all resources. Note: In libarchive 1.x, this function was declared to return *void*, which made it impossible to detect certain errors when **archive_read_close**() was invoked implicitly from this function.

The declaration is corrected beginning with libarchive 2.0.

Note that the library determines most of the relevant information about the archive by inspection. In particular, it automatically detects `gzip`(1) or `bzip2`(1) compression and transparently performs the appropriate decompression. It also automatically detects the archive format.

A complete description of the struct archive and struct archive_entry objects can be found in the overview manual page for `libarchive`(3).

CLIENT CALLBACKS

The callback functions must match the following prototypes:

typedef ssize_t **archive_read_callback**(*struct archive *, void *client_data, const void **buffer*)

typedef int **archive_skip_callback**(*struct archive *, void *client_data, size_t request*)

typedef int **archive_open_callback**(*struct archive *, void *client_data*)

typedef int **archive_close_callback**(*struct archive *, void *client_data*)

The open callback is invoked by **archive_open**(). It should return **ARCHIVE_OK** if the underlying file or data source is successfully opened. If the open fails, it should call **archive_set_error**() to register an error code and message and return **ARCHIVE_FATAL**.

The read callback is invoked whenever the library requires raw bytes from the archive. The read callback should read data into a buffer, set the `const void **buffer` argument to point to the available data, and return a count of the number of bytes available. The library will invoke the read callback again only after it has consumed this data. The library imposes no constraints on the size of the data blocks returned. On end-of-file, the read callback should return zero. On error, the read callback should invoke **archive_set_error**() to register an error code and message and return -1.

The skip callback is invoked when the library wants to ignore a block of data. The return value is the number of bytes actually skipped, which may differ from the request. If the callback cannot skip data, it should return zero. If the skip callback is not provided (the function pointer is `NULL`), the library will invoke the read function instead and simply discard the result. A skip callback can provide significant performance gains when reading uncompressed archives from slow disk drives or other media that can skip quickly.

The close callback is invoked by archive_close when the archive processing is complete. The callback should return **ARCHIVE_OK** on success. On failure, the callback should invoke **archive_set_error**() to register an error code and message and return **ARCHIVE_FATAL.**

EXAMPLE

The following illustrates basic usage of the library. In this example, the callback functions are simply wrappers around the standard `open`(2), `read`(2), and `close`(2) system calls.

```
void
list_archive(const char *name)
{
  struct mydata *mydata;
  struct archive *a;
  struct archive_entry *entry;

  mydata = malloc(sizeof(struct mydata));
  a = archive_read_new();
```

```
  mydata->name = name;
  archive_read_support_compression_all(a);
  archive_read_support_format_all(a);
  archive_read_open(a, mydata, myopen, myread, myclose);
  while (archive_read_next_header(a, &entry) == ARCHIVE_OK) {
    printf("%s\n",archive_entry_pathname(entry));
    archive_read_data_skip(a);
  }
  archive_read_finish(a);
  free(mydata);
}

ssize_t
myread(struct archive *a, void *client_data, const void **buff)
{
  struct mydata *mydata = client_data;

  *buff = mydata->buff;
  return (read(mydata->fd, mydata->buff, 10240));
}

int
myopen(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  mydata->fd = open(mydata->name, O_RDONLY);
  return (mydata->fd >= 0 ? ARCHIVE_OK : ARCHIVE_FATAL);
}

int
myclose(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  if (mydata->fd > 0)
    close(mydata->fd);
  return (ARCHIVE_OK);
}
```

RETURN VALUES

Most functions return zero on success, non-zero on error. The possible return codes include: **ARCHIVE_OK** (the operation succeeded), **ARCHIVE_WARN** (the operation succeeded but a non-critical error was encountered), **ARCHIVE_EOF** (end-of-archive was encountered), **ARCHIVE_RETRY** (the operation failed but can be retried), and **ARCHIVE_FATAL** (there was a fatal error; the archive should be closed immediately). Detailed error codes and textual descriptions are available from the **archive_errno**() and **archive_error_string**() functions.

archive_read_new() returns a pointer to a freshly allocated struct archive object. It returns NULL on error.

archive_read_data() returns a count of bytes actually read or zero at the end of the entry. On error, a value of **ARCHIVE_FATAL**, **ARCHIVE_WARN**, or **ARCHIVE_RETRY** is returned and an error code and textual description can be retrieved from the **archive_errno**() and **archive_error_string**() functions.

The library expects the client callbacks to behave similarly. If there is an error, you can use **archive_set_error**() to set an appropriate error code and description, then return one of the non-zero values above. (Note that the value eventually returned to the client may not be the same; many errors that are not critical at the level of basic I/O can prevent the archive from being properly read, thus most I/O errors eventually cause **ARCHIVE_FATAL** to be returned.)

SEE ALSO

tar(1), archive(3), archive_util(3), tar(5)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

BUGS

Many traditional archiver programs treat empty files as valid empty archives. For example, many implementations of tar(1) allow you to append entries to an empty file. Of course, it is impossible to determine the format of an empty file by inspecting the contents, so this library treats empty files as having a special “empty” format.

NAME

archive_clear_error, **archive_compression**, **archive_compression_name**, **archive_copy_error**, **archive_errno**, **archive_error_string**, **archive_format**, **archive_format_name**, **archive_set_error** — libarchive utility functions

SYNOPSIS

```
#include <archive.h>

void
archive_clear_error(struct archive *);

int
archive_compression(struct archive *);

const char *
archive_compression_name(struct archive *);

void
archive_copy_error(struct archive *, struct archive *);

int
archive_errno(struct archive *);

const char *
archive_error_string(struct archive *);

int
archive_format(struct archive *);

const char *
archive_format_name(struct archive *);

void
archive_set_error(struct archive *,int error_code,const char *fmt, ...);
```

DESCRIPTION

These functions provide access to various information about the struct archive object used in the libarchive(3) library.

archive_clear_error()
: Clears any error information left over from a previous call. Not generally used in client code.

archive_compression()
: Returns a numeric code indicating the current compression. This value is set by **archive_read_open**().

archive_compression_name()
: Returns a text description of the current compression suitable for display.

archive_copy_error()
: Copies error information from one archive to another.

archive_errno()
: Returns a numeric error code (see errno(2)) indicating the reason for the most recent error return.

archive_error_string()
: Returns a textual error message suitable for display. The error message here is usually more specific than that obtained from passing the result of **archive_errno**() to strerror(3).

archive_format()
: Returns a numeric code indicating the format of the current archive entry. This value is set by a successful call to **archive_read_next_header**(). Note that it is common for this value to

change from entry to entry. For example, a tar archive might have several entries that utilize GNU tar extensions and several entries that do not. These entries will have different format codes.

archive_format_name()
A textual description of the format of the current entry.

archive_set_error()
Sets the numeric error code and error description that will be returned by **archive_errno**() and **archive_error_string**(). This function should be used within I/O callbacks to set system-specific error codes and error descriptions. This function accepts a printf-like format string and arguments. However, you should be careful to use only the following printf format specifiers: “%c”, “%d”, “%jd”, “%jo”, “%ju”, “%jx”, “%ld”, “%lo”, “%lu”, “%lx”, “%o”, “%u”, “%s”, “%x”, “%%”. Field-width specifiers and other printf features are not uniformly supported and should not be used.

SEE ALSO

archive_read(3), archive_write(3), libarchive(3), printf(3)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

NAME

archive_write_new, **archive_write_set_format_cpio**, **archive_write_set_format_pax**, **archive_write_set_format_pax_restricted**, **archive_write_set_format_shar**, **archive_write_set_format_shar_binary**, **archive_write_set_format_ustar**, **archive_write_get_bytes_per_block**, **archive_write_set_bytes_per_block**, **archive_write_set_bytes_in_last_block**, **archive_write_set_compression_bzip2**, **archive_write_set_compression_compress**, **archive_write_set_compression_gzip**, **archive_write_set_compression_none**, **archive_write_set_compression_program**, **archive_write_open**, **archive_write_open_fd**, **archive_write_open_FILE**, **archive_write_open_filename**, **archive_write_open_memory**, **archive_write_header**, **archive_write_data**, **archive_write_finish_entry**, **archive_write_close**, **archive_write_finish** — functions for creating archives

SYNOPSIS

```
#include <archive.h>

struct archive *
archive_write_new(void);

int
archive_write_get_bytes_per_block(struct archive *);

int
archive_write_set_bytes_per_block(struct archive *, int bytes_per_block);

int
archive_write_set_bytes_in_last_block(struct archive *, int);

int
archive_write_set_compression_bzip2(struct archive *);

int
archive_write_set_compression_compress(struct archive *);

int
archive_write_set_compression_gzip(struct archive *);

int
archive_write_set_compression_none(struct archive *);

int
archive_write_set_compression_program(struct archive *,const char * cmd);

int
archive_write_set_format_cpio(struct archive *);

int
archive_write_set_format_pax(struct archive *);

int
archive_write_set_format_pax_restricted(struct archive *);

int
archive_write_set_format_shar(struct archive *);
```

int
archive_write_set_format_shar_binary(*struct archive **);

int
archive_write_set_format_ustar(*struct archive **);

int
archive_write_open(*struct archive **, *void *client_data*,
*archive_open_callback **, *archive_write_callback **,
*archive_close_callback **);

int
archive_write_open_fd(*struct archive **, *int fd*);

int
archive_write_open_FILE(*struct archive **, *FILE *file*);

int
archive_write_open_filename(*struct archive **, *const char *filename*);

int
archive_write_open_memory(*struct archive **, *void *buffer*,
size_t bufferSize, *size_t *outUsed*);

int
archive_write_header(*struct archive **, *struct archive_entry **);

ssize_t
archive_write_data(*struct archive **, *const void **, *size_t*);

int
archive_write_finish_entry(*struct archive **);

int
archive_write_close(*struct archive **);

int
archive_write_finish(*struct archive **);

DESCRIPTION

These functions provide a complete API for creating streaming archive files. The general process is to first create the struct archive object, set any desired options, initialize the archive, append entries, then close the archive and release all resources. The following summary describes the functions in approximately the order they are ordinarily used:

archive_write_new()
: Allocates and initializes a struct archive object suitable for writing a tar archive.

archive_write_set_bytes_per_block()
: Sets the block size used for writing the archive data. Every call to the write callback function, except possibly the last one, will use this value for the length. The third parameter is a boolean that specifies whether or not the final block written will be padded to the full block size. If it is zero, the last block will not be padded. If it is non-zero, padding will be added both before and after compression. The default is to use a block size of 10240 bytes and to pad the last block. Note that a block size of zero will suppress internal blocking and cause writes to be sent directly to the write callback as they occur.

archive_write_get_bytes_per_block()
Retrieve the block size to be used for writing. A value of -1 here indicates that the library should use default values. A value of zero indicates that internal blocking is suppressed.

archive_write_set_bytes_in_last_block()
Sets the block size used for writing the last block. If this value is zero, the last block will be padded to the same size as the other blocks. Otherwise, the final block will be padded to a multiple of this size. In particular, setting it to 1 will cause the final block to not be padded. For compressed output, any padding generated by this option is applied only after the compression. The uncompressed data is always unpadded. The default is to pad the last block to the full block size (note that **archive_write_open_filename**() will set this based on the file type). Unlike the other "set" functions, this function can be called after the archive is opened.

archive_write_get_bytes_in_last_block()
Retrieve the currently-set value for last block size. A value of -1 here indicates that the library should use default values.

archive_write_set_format_cpio(), **archive_write_set_format_pax**(), **archive_write_set_format_pax_restricted**(), **archive_write_set_format_shar**(), **archive_write_set_format_shar_binary**(), **archive_write_set_format_ustar**()
Sets the format that will be used for the archive. The library can write POSIX octet-oriented cpio format archives, POSIX-standard "pax interchange" format archives, traditional "shar" archives, enhanced "binary" shar archives that store a variety of file attributes and handle binary files, and POSIX-standard "ustar" archives. The pax interchange format is a backwards-compatible tar format that adds key/value attributes to each entry and supports arbitrary filenames, linknames, uids, sizes, etc. "Restricted pax interchange format" is the library default; this is the same as pax format, but suppresses the pax extended header for most normal files. In most cases, this will result in ordinary ustar archives.

archive_write_set_compression_bzip2(), **archive_write_set_compression_compress**(), **archive_write_set_compression_gzip**(), **archive_write_set_compression_none**()
The resulting archive will be compressed as specified. Note that the compressed output is always properly blocked.

archive_write_set_compression_program()
The archive will be fed into the specified compression program. The output of that program is blocked and written to the client write callbacks.

archive_write_open()
Freeze the settings, open the archive, and prepare for writing entries. This is the most generic form of this function, which accepts pointers to three callback functions which will be invoked by the compression layer to write the constructed archive.

archive_write_open_fd()
A convenience form of **archive_write_open**() that accepts a file descriptor. The **archive_write_open_fd**() function is safe for use with tape drives or other block-oriented devices.

archive_write_open_FILE()
A convenience form of **archive_write_open**() that accepts a *FILE* * pointer. Note that **archive_write_open_FILE**() is not safe for writing to tape drives or other devices that require correct blocking.

archive_write_open_file()
A deprecated synonym for **archive_write_open_filename**().

archive_write_open_filename()
A convenience form of **archive_write_open**() that accepts a filename. A NULL argument indicates that the output should be written to standard output; an argument of "-" will open a file with that name. If you have not invoked **archive_write_set_bytes_in_last_block**(), then **archive_write_open_filename**() will adjust the last-block padding depending on the file: it will enable padding when writing to standard output or to a character or block device node, it will disable padding otherwise. You can override this by manually invoking **archive_write_set_bytes_in_last_block**() before calling **archive_write_open**(). The **archive_write_open_filename**() function is safe for use with tape drives or other block-oriented devices.

archive_write_open_memory()
A convenience form of **archive_write_open**() that accepts a pointer to a block of memory that will receive the archive. The final *size_t* * argument points to a variable that will be updated after each write to reflect how much of the buffer is currently in use. You should be careful to ensure that this variable remains allocated until after the archive is closed.

archive_write_header()
Build and write a header using the data in the provided struct archive_entry structure. See archive_entry(3) for information on creating and populating struct archive_entry objects.

archive_write_data()
Write data corresponding to the header just written. Returns number of bytes written or -1 on error.

archive_write_finish_entry()
Close out the entry just written. In particular, this writes out the final padding required by some formats. Ordinarily, clients never need to call this, as it is called automatically by **archive_write_next_header**() and **archive_write_close**() as needed.

archive_write_close()
Complete the archive and invoke the close callback.

archive_write_finish()
Invokes **archive_write_close**() if it was not invoked manually, then releases all resources. Note that this function was declared to return *void* in libarchive 1.x, which made it impossible to detect errors when **archive_write_close**() was invoked implicitly from this function. This is corrected beginning with libarchive 2.0.

More information about the *struct archive* object and the overall design of the library can be found in the libarchive(3) overview.

IMPLEMENTATION

Compression support is built-in to libarchive, which uses zlib and bzlib to handle gzip and bzip2 compression, respectively.

CLIENT CALLBACKS

To use this library, you will need to define and register callback functions that will be invoked to write data to the resulting archive. These functions are registered by calling **archive_write_open**():

typedef int **archive_open_callback**(*struct archive **, *void *client_data*)

The open callback is invoked by **archive_write_open**(). It should return **ARCHIVE_OK** if the underlying file or data source is successfully opened. If the open fails, it should call **archive_set_error**() to

register an error code and message and return **ARCHIVE_FATAL**.

typedef ssize_t **archive_write_callback**(*struct archive *, void *client_data, void *buffer, size_t length*)

The write callback is invoked whenever the library needs to write raw bytes to the archive. For correct blocking, each call to the write callback function should translate into a single `write`(2) system call. This is especially critical when writing archives to tape drives. On success, the write callback should return the number of bytes actually written. On error, the callback should invoke **archive_set_error**() to register an error code and message and return -1.

typedef int **archive_close_callback**(*struct archive *, void *client_data*)

The close callback is invoked by archive_close when the archive processing is complete. The callback should return **ARCHIVE_OK** on success. On failure, the callback should invoke **archive_set_error**() to register an error code and message and return **ARCHIVE_FATAL.**

EXAMPLE

The following sketch illustrates basic usage of the library. In this example, the callback functions are simply wrappers around the standard `open`(2), `write`(2), and `close`(2) system calls.

```
#include <sys/stat.h>
#include <archive.h>
#include <archive_entry.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

struct mydata {
        const char *name;
        int fd;
};

int
myopen(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  mydata->fd = open(mydata->name, O_WRONLY | O_CREAT, 0644);
  if (mydata->fd >= 0)
    return (ARCHIVE_OK);
  else
    return (ARCHIVE_FATAL);
}

ssize_t
mywrite(struct archive *a, void *client_data, void *buff, size_t n)
{
  struct mydata *mydata = client_data;

  return (write(mydata->fd, buff, n));
}
```

```
int
myclose(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  if (mydata->fd > 0)
    close(mydata->fd);
  return (0);
}

void
write_archive(const char *outname, const char **filename)
{
  struct mydata *mydata = malloc(sizeof(struct mydata));
  struct archive *a;
  struct archive_entry *entry;
  struct stat st;
  char buff[8192];
  int len;
  int fd;

  a = archive_write_new();
  mydata->name = outname;
  archive_write_set_compression_gzip(a);
  archive_write_set_format_ustar(a);
  archive_write_open(a, mydata, myopen, mywrite, myclose);
  while (*filename) {
    stat(*filename, &st);
    entry = archive_entry_new();
    archive_entry_copy_stat(entry, &st);
    archive_entry_set_pathname(entry, *filename);
    archive_write_header(a, entry);
    fd = open(*filename, O_RDONLY);
    len = read(fd, buff, sizeof(buff));
    while ( len > 0 ) {
        archive_write_data(a, buff, len);
        len = read(fd, buff, sizeof(buff));
    }
    archive_entry_free(entry);
    filename++;
  }
  archive_write_finish(a);
}

int main(int argc, const char **argv)
{
        const char *outname;
        argv++;
        outname = argv++;
        write_archive(outname, argv);
        return 0;
}
```

RETURN VALUES

Most functions return **ARCHIVE_OK** (zero) on success, or one of several non-zero error codes for errors. Specific error codes include: **ARCHIVE_RETRY** for operations that might succeed if retried, **ARCHIVE_WARN** for unusual conditions that do not prevent further operations, and **ARCHIVE_FATAL** for serious errors that make remaining operations impossible. The **archive_errno**() and **archive_error_string**() functions can be used to retrieve an appropriate error code and a textual error message.

archive_write_new() returns a pointer to a newly-allocated struct archive object.

archive_write_data() returns a count of the number of bytes actually written. On error, -1 is returned and the **archive_errno**() and **archive_error_string**() functions will return appropriate values. Note that if the client-provided write callback function returns a non-zero value, that error will be propagated back to the caller through whatever API function resulted in that call, which may include **archive_write_header**(), **archive_write_data**(), **archive_write_close**(), or **archive_write_finish**(). The client callback can call **archive_set_error**() to provide values that can then be retrieved by **archive_errno**() and **archive_error_string**().

SEE ALSO

tar(1), libarchive(3), tar(5)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

BUGS

There are many peculiar bugs in historic tar implementations that may cause certain programs to reject archives written by this library. For example, several historic implementations calculated header checksums incorrectly and will thus reject valid archives; GNU tar does not fully support pax interchange format; some old tar implementations required specific field terminations.

The default pax interchange format eliminates most of the historic tar limitations and provides a generic key/value attribute facility for vendor-defined extensions. One oversight in POSIX is the failure to provide a standard attribute for large device numbers. This library uses “SCHILY.devminor” and “SCHILY.devmajor” for device numbers that exceed the range supported by the backwards-compatible ustar header. These keys are compatible with Joerg Schilling’s **star** archiver. Other implementations may not recognize these keys and will thus be unable to correctly restore device nodes with large device numbers from archives created by this library.

NAME

archive_write_disk_new, **archive_write_disk_set_options**, **archive_write_disk_set_skip_file**, **archive_write_disk_set_group_lookup**, **archive_write_disk_set_standard_lookup**, **archive_write_disk_set_user_lookup**, **archive_write_header**, **archive_write_data**, **archive_write_finish_entry**, **archive_write_close**, **archive_write_finish** — functions for creating objects on disk

SYNOPSIS

```
#include <archive.h>

struct archive *
archive_write_disk_new(void);

int
archive_write_disk_set_options(struct archive *, int flags);

int
archive_write_disk_set_skip_file(struct archive *, dev_t, ino_t);

int
archive_write_disk_set_group_lookup(struct archive *,void *,
    gid_t (*)(void *, const char *gname, gid_t gid),
    void (*cleanup)(void *) );

int
archive_write_disk_set_standard_lookup(struct archive *);

int
archive_write_disk_set_user_lookup(struct archive *,void *,
    uid_t (*)(void *, const char *uname, uid_t uid),
    void (*cleanup)(void *) );

int
archive_write_header(struct archive *, struct archive_entry *);

ssize_t
archive_write_data(struct archive *, const void *, size_t);

int
archive_write_finish_entry(struct archive *);

int
archive_write_close(struct archive *);

int
archive_write_finish(struct archive *);
```

DESCRIPTION

These functions provide a complete API for creating objects on disk from struct archive_entry descriptions. They are most naturally used when extracting objects from an archive using the **archive_read**() interface. The general process is to read struct archive_entry objects from an archive, then write those objects to a struct archive object created using the **archive_write_disk**() family functions. This interface is deliberately very similar to the **archive_write**() interface used to write objects to a streaming archive.

archive_write_disk_new()

Allocates and initializes a struct archive object suitable for writing objects to disk.

archive_write_disk_set_skip_file()
Records the device and inode numbers of a file that should not be overwritten. This is typically used to ensure that an extraction process does not overwrite the archive from which objects are being read. This capability is technically unnecessary but can be a significant performance optimization in practice.

archive_write_disk_set_options()
The options field consists of a bitwise OR of one or more of the following values:

ARCHIVE_EXTRACT_OWNER
The user and group IDs should be set on the restored file. By default, the user and group IDs are not restored.

ARCHIVE_EXTRACT_PERM
Full permissions (including SGID, SUID, and sticky bits) should be restored exactly as specified, without obeying the current umask. Note that SUID and SGID bits can only be restored if the user and group ID of the object on disk are correct. If **ARCHIVE_EXTRACT_OWNER** is not specified, then SUID and SGID bits will only be restored if the default user and group IDs of newly-created objects on disk happen to match those specified in the archive entry. By default, only basic permissions are restored, and umask is obeyed.

ARCHIVE_EXTRACT_TIME
The timestamps (mtime, ctime, and atime) should be restored. By default, they are ignored. Note that restoring of atime is not currently supported.

ARCHIVE_EXTRACT_NO_OVERWRITE
Existing files on disk will not be overwritten. By default, existing regular files are truncated and overwritten; existing directories will have their permissions updated; other pre-existing objects are unlinked and recreated from scratch.

ARCHIVE_EXTRACT_UNLINK
Existing files on disk will be unlinked before any attempt to create them. In some cases, this can prove to be a significant performance improvement. By default, existing files are truncated and rewritten, but the file is not recreated. In particular, the default behavior does not break existing hard links.

ARCHIVE_EXTRACT_ACL
Attempt to restore ACLs. By default, extended ACLs are ignored.

ARCHIVE_EXTRACT_FFLAGS
Attempt to restore extended file flags. By default, file flags are ignored.

ARCHIVE_EXTRACT_XATTR
Attempt to restore POSIX.1e extended attributes. By default, they are ignored.

ARCHIVE_EXTRACT_SECURE_SYMLINKS
Refuse to extract any object whose final location would be altered by a symlink on disk. This is intended to help guard against a variety of mischief caused by archives that (deliberately or otherwise) extract files outside of the current directory. The default is not to perform this check. If **ARCHIVE_EXTRACT_UNLINK** is specified together with this option, the library will remove any intermediate symlinks it finds and return an error only if such symlink could not be removed.

ARCHIVE_EXTRACT_SECURE_NODOTDOT
Refuse to extract a path that contains a `..` element anywhere within it. The default is to not refuse such paths. Note that paths ending in `..` always cause an error, regardless of this flag.

ARCHIVE_EXTRACT_SPARSE
Scan data for blocks of NUL bytes and try to recreate them with holes. This results in sparse files, independent of whether the archive format supports or uses them.

archive_write_disk_set_group_lookup(), **archive_write_disk_set_user_lookup**()
The struct archive_entry objects contain both names and ids that can be used to identify users and groups. These names and ids describe the ownership of the file itself and also appear in ACL lists. By default, the library uses the ids and ignores the names, but this can be overridden by registering user and group lookup functions. To register, you must provide a lookup function which accepts both a name and id and returns a suitable id. You may also provide a void * pointer to a private data structure and a cleanup function for that data. The cleanup function will be invoked when the struct archive object is destroyed.

archive_write_disk_set_standard_lookup()
This convenience function installs a standard set of user and group lookup functions. These functions use `getpwnam`(3) and `getgrnam`(3) to convert names to ids, defaulting to the ids if the names cannot be looked up. These functions also implement a simple memory cache to reduce the number of calls to `getpwnam`(3) and `getgrnam`(3).

archive_write_header()
Build and write a header using the data in the provided struct archive_entry structure. See `archive_entry`(3) for information on creating and populating struct archive_entry objects.

archive_write_data()
Write data corresponding to the header just written. Returns number of bytes written or -1 on error.

archive_write_finish_entry()
Close out the entry just written. Ordinarily, clients never need to call this, as it is called automatically by **archive_write_next_header**() and **archive_write_close**() as needed.

archive_write_close()
Set any attributes that could not be set during the initial restore. For example, directory timestamps are not restored initially because restoring a subsequent file would alter that timestamp. Similarly, non-writable directories are initially created with write permissions (so that their contents can be restored). The **archive_write_disk_new** library maintains a list of all such deferred attributes and sets them when this function is invoked.

archive_write_finish()
Invokes **archive_write_close**() if it was not invoked manually, then releases all resources.

More information about the *struct archive* object and the overall design of the library can be found in the `libarchive`(3) overview. Many of these functions are also documented under `archive_write`(3).

RETURN VALUES

Most functions return **ARCHIVE_OK** (zero) on success, or one of several non-zero error codes for errors. Specific error codes include: **ARCHIVE_RETRY** for operations that might succeed if retried, **ARCHIVE_WARN** for unusual conditions that do not prevent further operations, and **ARCHIVE_FATAL** for serious errors that make remaining operations impossible. The **archive_errno**() and **archive_error_string**() functions can be used to retrieve an appropriate error code and a textual error message.

archive_write_disk_new() returns a pointer to a newly-allocated struct archive object.

archive_write_data() returns a count of the number of bytes actually written. On error, -1 is returned and the **archive_errno**() and **archive_error_string**() functions will return appropriate values.

SEE ALSO

`archive_read`(3), `archive_write`(3), `tar`(1), `libarchive`(3)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3. The **archive_write_disk** interface was added to **libarchive 2.0** and first appeared in FreeBSD 6.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

BUGS

Directories are actually extracted in two distinct phases. Directories are created during **archive_write_header**(), but final permissions are not set until **archive_write_close**(). This separation is necessary to correctly handle borderline cases such as a non-writable directory containing files, but can cause unexpected results. In particular, directory permissions are not fully restored until the archive is closed. If you use `chdir`(2) to change the current directory between calls to **archive_read_extract**() or before calling **archive_read_close**(), you may confuse the permission-setting logic with the result that directory permissions are restored incorrectly.

The library attempts to create objects with filenames longer than **PATH_MAX** by creating prefixes of the full path and changing the current directory. Currently, this logic is limited in scope; the fixup pass does not work correctly for such objects and the symlink security check option disables the support for very long pathnames.

Restoring the path `aa/../bb` does create each intermediate directory. In particular, the directory `aa` is created as well as the final object `bb`. In theory, this can be exploited to create an entire directory heirarchy with a single request. Of course, this does not work if the **ARCHIVE_EXTRACT_NODOTDOT** option is specified.

Implicit directories are always created obeying the current umask. Explicit objects are created obeying the current umask unless **ARCHIVE_EXTRACT_PERM** is specified, in which case they current umask is ignored.

SGID and SUID bits are restored only if the correct user and group could be set. If **ARCHIVE_EXTRACT_OWNER** is not specified, then no attempt is made to set the ownership. In this case, SGID and SUID bits are restored only if the user and group of the final object happen to match those specified in the entry.

The “standard” user-id and group-id lookup functions are not the defaults because `getgrnam`(3) and `getpwnam`(3) are sometimes too large for particular applications. The current design allows the application author to use a more compact implementation when appropriate.

There should be a corresponding **archive_read_disk** interface that walks a directory heirarchy and returns archive entry objects.

NAME

ar_answer, ar_close, ar_delete, ar_gethostbyname, ar_gethostbyaddr, ar_init, ar_open, ar_timeout - Asynchronous DNS library routines

SYNOPSIS

#include arlib.h

struct hostent *ar_answer(dataptr, size)
char *dataptr;
int size;

void ar_close();

int ar_delete(dataptr, size)
char *dataptr;
int size;

int ar_gethostbyname(name, dataptr, size)
char *name;
char *dataptr;
int size;

int ar_gethostbyaddr(name, dataptr, size)
char *name;
char *dataptr;
int size;

int ar_init(flags)
int flags;

int ar_open();

long ar_timeout(time, dataptr, size)
long time;
char *dataptr;
int size;

DESCRIPTION

This small library of DNS routines is intended to provide an asynchronous interface to performing hostname and IP number lookups. Only lookups of Internet domain are handled as yet. To use this set of routines properly, the presence of the **BIND 4.8** resolve libraries is required (or any library derived from it).

This library should be used in conjunction with **select(2)** to wait for the name server's reply to arrive or the lookup to timeout.

To open a fd for talking to the name server, either **ar_open()** or ar_init() must be used. **ar_open()** will open either a datagram socket or a virtual circuit with the name server, depending on the flags set in the _res structure (see **resolv(5)**). In both cases, if the socket

> i **ar_init()** is used to both open the socket (as in **ar_open()**) and initialize the queues used by this library. The values recognized as parameters to **ar_init()** are:

```
#define ARES_INITLIST   1
#define ARES_CALLINIT   2
#define ARES_INITSOCK   4
#define ARES_INITDEBG   8
```

ARES_INITLIST initializes the list of queries waiting for replies. ARES_CALLINIT is a flag which when set causes **res_init()** to be called. ARES_INITSOCK will close the current socket if it is open and call **ar_open()** to open a new one, returning the fd for that socket. ARES_INITDEBG sets the RES_DEBUG flag of the **_res** structure. ARES_INITCACH is as yet, unused and is for future use where the library keeps its own cache of replies.

To send a query about either a hostname or an IP number, **ar_gethostbyname()** and **ar_gethostbyaddr()** must be used. Each takes either a pointer to the hostname or the IP number respectively for use when making the query. In addition to this, both (optionally) can be passed a pointer to data, dataptr, with the size also passed which can be used for identifying individual queries. A copy of the area pointed to is made if dataptr is non NULL and size is non zero. These functions will always return NULL unless the answer to the query is found in internal caches. A new flag, RES_CHECKPTR is checked during the processing of answers for **ar_gethostbyname()** which will automatically cause a reverse lookup to be queued, causing a failure if that reply differs from the original.

To check for a query, **ar_answer()** is called with a pointer to an area of memory which is sufficient to hold what was originally passed via **ar_gethostbyname()** or **ar_gethostbyaddr()** through dataptr. If an answer is found, a pointer to the host information is returned and the data segment copied if dataptr is non NULL and it was originally passed. The size of the copied data is the smaller of the passed size and that of original data stored.

To expire old queries, **ar_timeout()** is called with the 'current' time (or the time for which you want to do timeouts for). If a queue entry is too old, it will be expired when it has exhausted all available avenues for lookups and the data segment for the expired query copied into dataptr. The size of the copied data is the smaller of the passed size and that of the original stored data. Only 1 entry is thus expired with each call, requiring that it be called immediately after an expiration to check for others. In addition to expiring lookups, **ar_timeout()** also triggers resends of queries and the searching of the domain tree for the host, the latter works from the **_res** structure of **resolv(5).**

To delete entries from the queue, **ar_delete()** can be used and by passing the pointer and size of the data segment, all queries have their data segments checked (if present) for an exact match, being deleted if and only if there is a match. A NULL pointer passed to ar_deleted() matches all queries which were called with a NULL dataptr parameter. The amount of data compared is the smaller of the size passed and that of the data stored for the queue entry being compared.

To close a socket opened by **ar_open()** , **ar_close()** should be used so that it is closed and also marked closed within this library.

DIAGNOSIS

ar_open() returns -1 if a socket isn't open and could not be opened; otherwise returns the current fd open or the fd it opened.

ar_init() returns -1 for any errors, the value returned by **res_init()** if **res_init()** was called, the return value for **ar_open()** if that was called or the current socket open if 0 is passed and a socket is open. If neither **res_init()** or **ar_open()** are called and the flags are non-zero, -2 is returned.

ar_gethostbyaddr() and **ar_gethostbyname()** will always return NULL in this version but may return a pointer to a hostent structure if a cache is being used and the answer is found in the cache.

ar_answer() returns NULL if the answer is either not found or the query returned an error and another attempt at a lookup is attempted. If an answer was found, it returned a pointer to this structure and the contents of the data segment copied over.

ar_timeout() returns the time when it should be called next or 0 if there are no queries in the queue to be checked later. If any queries are expired, the data segment is copied over if dataptr is non NULL.

ar_delete() returns the number of entries that were found to match and consequently deleted.

SEE ALSO

gethostbyaddr(3), gethostbyname(3), resolv(5)

FILES

arlib.h
/usr/include/resolv.h
/usr/include/arpa/nameser.h
/etc/resolv.conf

BUGS

The results of a successful call to ar_answer() destroy the structure for any previous calls.

AUTHOR

Darren Reed. Email address: avalon@coombs.anu.edu.au

NAME

asin, **asinf** — arc sine function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
asin(double x);

float
asinf(float x);
```

DESCRIPTION

The **asin**() and **asinf**() functions compute the principal value of the arc sine of *x* in the range [-π/2, +π/2].

RETURN VALUES

If |x|>1, **asin**(*x*) and **asinf**(*x*) return NaN and set the global variable *errno* to EDOM.

SEE ALSO

acos(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **asin**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

asinh, **asinhf** — inverse hyperbolic sine function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
asinh(double x);

float
asinhf(float x);
```

DESCRIPTION

The **asinh**() and **asinhf**() functions compute the inverse hyperbolic sine of the real argument

RETURN VALUES

The **asinh**() and **asinhf**() functions return the inverse hyperbolic sine of *x*.

SEE ALSO

acosh(3), atanh(3), exp(3), math(3)

HISTORY

The **asinh**() function appeared in 4.3BSD.

NAME

assert — expression verification macro

SYNOPSIS

```
#include <assert.h>

assert(expression);
```

DESCRIPTION

The **assert**() macro tests the given *expression* and if it is false, the calling process is terminated. A diagnostic message, consisting of the text of the expression, the name of the source file, the line number and the enclosing function, is written to *stderr* and the abort(3) function is called, effectively terminating the program.

If *expression* is true, the **assert**() macro does nothing.

The **assert**() macro may be removed at compile time with the cc(1) option **-DNDEBUG**.

DIAGNOSTICS

The following diagnostic message is written to *stderr* if *expression* is false:

```
"assertion \"%s\" failed: file \"%s\", line %d, function \"%s\"\n", \
                    "expression", __FILE__, __LINE__, __func__);
```

SEE ALSO

cc(1), _DIAGASSERT(3), abort(3)

STANDARDS

The **assert**() macro conforms to ISO/IEC 9899:1999 (“ISO C99”).

HISTORY

A **assert** macro appeared in Version 6 AT&T UNIX.

Information on the name of the enclosing function appeared in ISO/IEC 9899:1999 (“ISO C99”).

NAME

atan, **atanf** — arc tangent function of one variable

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
atan(double x);

float
atanf(float x);
```

DESCRIPTION

The **atan**() and **atanf**() functions compute the principal value of the arc tangent of *x* in the range [$-\pi/2$, $+\pi/2$].

SEE ALSO

acos(3), asin(3), atan2(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **atan**() functions conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

atan2, **atan2f** — arc tangent function of two variables

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
atan2(double y, double x);

float
atan2f(float y, float x);
```

DESCRIPTION

The **atan2**() and **atan2f**() functions compute the principal value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value.

RETURN VALUES

The **atan2**() function, if successful, returns the arc tangent of y/x in the range $[-\pi, +\pi]$ radians. If both x and y are zero, the global variable *errno* is set to EDOM. On the VAX:

atan2(y, x) :=	**atan**(y/x)	if $x > 0$,
	sign(y)*(π - **atan**(|y/x|))	if $x < 0$,
	0	if x = y = 0, or
	sign(y)*π/2	if $x = 0$ y.

NOTES

The function **atan2**() defines "if x > 0," **atan2**(0, 0) = 0 on a VAX despite that previously **atan2**(0, 0) may have generated an error message. The reasons for assigning a value to **atan2**(0, 0) are these:

1. Programs that test arguments to avoid computing **atan2**(0, 0) must be indifferent to its value. Programs that require it to be invalid are vulnerable to diverse reactions to that invalidity on diverse computer systems.

2. The **atan2**() function is used mostly to convert from rectangular (x,y) to polar (r,θ) coordinates that must satisfy x = r*cosθ and y = r*sinθ. These equations are satisfied when (x=0,y=0) is mapped to (r=0,θ=0) on a VAX. In general, conversions to polar coordinates should be computed thus:

   r := hypot(x,y); ... := $\sqrt{(x^2+y^2)}$
   θ := atan2(y,x).

3. The foregoing formulas need not be altered to cope in a reasonable way with signed zeros and infinities on a machine that conforms to IEEE 754; the versions of hypot(3) and **atan2**() provided for such a machine are designed to handle all cases. That is why **atan2**(± 0, -0) = $\pm\pi$ for instance. In general the formulas above are equivalent to these:

   r := $\sqrt{(x*x+y*y)}$; if r = 0 then x := copysign(1,x);

SEE ALSO

acos(3), asin(3), atan(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **atan2**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

atanh, **atanhf** — inverse hyperbolic tangent function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
atanh(double x);

float
atanhf(float x);
```

DESCRIPTION

The **atanh**() and **atanhf**() functions compute the inverse hyperbolic tangent of the real argument *x*.

RETURN VALUES

If |x|≥1, **atanh**(*x*) and **atanhf**(*x*) return +inf, -inf or NaN, and sets the global variable *errno* to EDOM.

SEE ALSO

acosh(3), asinh(3), exp(3), math(3)

HISTORY

The **atanh**() function appeared in 4.3BSD.

NAME

atexit — register a function to be called on exit

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
atexit(void (*function)(void));
```

DESCRIPTION

The **atexit**() function registers the given *function* to be called at program exit, whether via exit(3) or via return from the program's *main*. Functions so registered are called in reverse order; no arguments are passed. At least 32 functions can always be registered, and more are allowed as long as sufficient memory can be allocated.

RETURN VALUES

The **atexit**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[ENOMEM] No memory was available to add the function to the list. The existing list of functions is unmodified.

SEE ALSO

exit(3)

STANDARDS

The **atexit**() function conforms to ANSI X3.159-1989 ("ANSI C89").

NAME

ATF_ADD_TEST_CASE, **ATF_CHECK**, **ATF_CHECK_EQUAL**, **ATF_CHECK_THROW**, **ATF_FAIL**, **ATF_INIT_TEST_CASES**, **ATF_PASS**, **ATF_SKIP**, **ATF_TEST_CASE**, **ATF_TEST_CASE_BODY**, **ATF_TEST_CASE_CLEANUP**, **ATF_TEST_CASE_HEAD**, **ATF_TEST_CASE_WITH_CLEANUP** — C++ API to write ATF-based test programs

SYNOPSIS

#include <atf-c++.hpp>

ATF_ADD_TEST_CASE(*tcs*, *name*);

ATF_CHECK(*expression*);

ATF_CHECK_EQUAL(*expression_1*, *expression_2*);

ATF_CHECK_THROW(*statement_1*, *expected_exception*);

ATF_FAIL(*reason*);

ATF_INIT_TEST_CASES(*tcs*);

ATF_PASS();

ATF_SKIP(*reason*);

ATF_TEST_CASE(*name*);

ATF_TEST_CASE_BODY(*name*);

ATF_TEST_CASE_CLEANUP(*name*);

ATF_TEST_CASE_HEAD(*name*);

ATF_TEST_CASE_WITH_CLEANUP(*name*);

DESCRIPTION

ATF provides a mostly-macro-based programming interface to implement test programs in C or C++. This interface is backed by a C++ implementation, but this fact is hidden from the developer as much as possible through the use of macros to simplify programming. However, the use of C++ is not hidden everywhere and while you can implement test cases without knowing anything at all about the object model underneath the provided calls, you might need some minimum notions of the language in very specific circumstances.

C++-based test programs always follow this template:

```
extern "C" {
... C-specific includes go here ...
}

... C++-specific includes go here ...

#include <atf-c++.hpp>

ATF_TEST_CASE(tc1);
ATF_TEST_CASE_HEAD(tc1)
{
   ... first test case's header ...
}
ATF_TEST_CASE_BODY(tc1)
{
```

```
    ... first test case's body ...
}

ATF_TEST_CASE_WITH_CLEANUP(tc2);
ATF_TEST_CASE_HEAD(tc2)
{
    ... second test case's header ...
}
ATF_TEST_CASE_BODY(tc2)
{
    ... second test case's body ...
}
ATF_TEST_CASE_CLEANUP(tc2)
{
    ... second test case's cleanup ...
}

... additional test cases ...

ATF_INIT_TEST_CASES(tcs)
{
    ATF_ADD_TEST_CASE(tcs, tc1)
    ATF_ADD_TEST_CASE(tcs, tc2)
    ... add additional test cases ...
}
```

Definition of test cases

Test cases have an identifier and are composed of three different parts: the header, the body and an optional cleanup routine, all of which are described in `atf-test-case`(8). To define test cases, one can use the **ATF_TEST_CASE**() or the **ATF_TEST_CASE_WITH_CLEANUP**() macros, which take a single parameter specifiying the test case's name. The former does not allow the specification of a cleanup routine for the test case while the latter does. It is important to note that these *do not* set the test case up for execution when the program is run. In order to do so, a later registration is needed through the **ATF_ADD_TEST_CASE**() macro detailed in **Program initialization**.

Later on, one must define the three parts of the body by means of three functions. Their headers are given by the **ATF_TEST_CASE_HEAD**(), **ATF_TEST_CASE_BODY**() and **ATF_TEST_CASE_CLEANUP**() macros, all of which take the test case's name. Following each of these, a block of code is expected, surrounded by the opening and closing brackets.

Program initialization

The library provides a way to easily define the test program's **main**() function. You should never define one on your own, but rely on the library to do it for you. This is done by using the **ATF_INIT_TEST_CASES**() macro, which is passed the name of the list that will hold the test cases. This name can be whatever you want as long as it is a valid variable value.

After the macro, you are supposed to provide the body of a function, which should only use the **ATF_ADD_TEST_CASE**() macro to register the test cases the test program will execute. The first parameter of this macro matches the name you provided in the former call.

Header definitions
The test case's header can define the meta-data by using the **set**() method, which takes two parameters: the first one specifies the meta-data variable to be set and the second one specifies its value. Both of them are strings.

Configuration variables
The test case has read-only access to the current configuration variables by means of the *bool* **has_config_var**() and the *std::string* **get_config_var**() methods, which can be called in any of the three parts of a test case.

Access to the source directory
It is possible to get the path to the test case's source directory from any of its three components by querying the 'srcdir' configuration variable.

Requiring programs
Aside from the *require.progs* meta-data variable available in the header only, one can also check for additional programs in the test case's body by using the **require_prog**() function, which takes the base name or full path of a single binary. Relative paths are forbidden. If it is not found, the test case will be automatically skipped.

Test case finalization
The test case finalizes either when the body reaches its end, at which point the test is assumed to have *passed*, or at any explicit call to **ATF_PASS**(), **ATF_FAIL**() or **ATF_SKIP**(). These three macros terminate the execution of the test case immediately. The cleanup routine will be processed afterwards in a completely automated way, regardless of the test case's termination reason.

ATF_PASS() does not take any parameters. **ATF_FAIL**() and **ATF_SKIP**() take a single string that describes why the test case failed or was skipped, respectively. It is very important to provide a clear error message in both cases so that the user can quickly know why the test did not pass.

Helper macros for common checks
The library provides several macros that are very handy in multiple situations. These basically check some condition after executing a given statement or processing a given expression and, if the condition is not met, they automatically call **ATF_FAIL**() with an appropriate error message.

ATF_CHECK() takes an expression and raises a failure if it evaluates to false.

ATF_CHECK_EQUAL() takes two expressions and raises a failure if the two do not evaluate to the same exact value.

ATF_CHECK_THROW() takes a statement and the name of an exception and raises a failure if the statement did not throw the specified exception.

EXAMPLES

The following shows a complete test program with a single test case that validates the addition operator:

```
#include <atf-c++.hpp>

ATF_TEST_CASE(addition);
ATF_TEST_CASE_HEAD(addition)
{
    set("descr", "Sample tests for the addition operator");
}
ATF_TEST_CASE_BODY(addition)
```

```
{
    ATF_CHECK_EQUAL(0 + 0, 0);
    ATF_CHECK_EQUAL(0 + 1, 1);
    ATF_CHECK_EQUAL(1 + 0, 1);

    ATF_CHECK_EQUAL(1 + 1, 2);

    ATF_CHECK_EQUAL(100 + 200, 300);
}

ATF_INIT_TEST_CASES(tcs)
{
    ATF_ADD_TEST_CASE(tcs, addition);
}
```

SEE ALSO

atf-test-program(1), atf(7), atf-test-case(8)

NAME

ATF_CHECK, **ATF_CHECK_EQUAL**, **ATF_TC**, **ATF_TC_BODY**, **ATF_TC_BODY_NAME**, **ATF_TC_CLEANUP**, **ATF_TC_CLEANUP_NAME**, **ATF_TC_HEAD**, **ATF_TC_HEAD_NAME**, **ATF_TC_NAME**, **ATF_TC_WITH_CLEANUP**, **ATF_TP_ADD_TC**, **ATF_TP_ADD_TCS**, **atf_no_error**, **atf_tc_fail**, **atf_tc_pass**, **atf_tc_skip** — C API to write ATF-based test programs

SYNOPSIS

#include <atf-c.h>

ATF_CHECK(*expression*);

ATF_CHECK_EQUAL(*expression_1*, *expression_2*);

ATF_TC(*name*);

ATF_TC_BODY(*name*);

ATF_TC_BODY_NAME(*name*);

ATF_TC_CLEANUP(*name*);

ATF_TC_CLEANUP_NAME(*name*);

ATF_TC_HEAD(*name*);

ATF_TC_HEAD_NAME(*name*);

ATF_TC_NAME(*name*);

ATF_TC_WITH_CLEANUP(*name*);

ATF_TP_ADD_TC(*tp_name*);

ATF_TP_ADD_TCS(*tp_name*, *tc_name*);

atf_no_error();

atf_tc_fail(*reason*);

atf_tc_pass();

atf_tc_skip(*reason*);

DESCRIPTION

The ATF

C-based test programs always follow this template:

```
. . . C-specific includes go here . . .

#include <atf-c.h>

ATF_TC(tc1);
ATF_TC_HEAD(tc1)
{
   ... first test case's header ...
}
ATF_TC_BODY(tc1)
{
   ... first test case's body ...
}
```

```
ATF_TC_WITH_CLEANUP(tc2);
ATF_TC_HEAD(tc2)
{
   ... second test case's header ...
}
ATF_TC_BODY(tc2)
{
   ... second test case's body ...
}
ATF_TC_CLEANUP(tc2)
{
   ... second test case's cleanup ...
}

... additional test cases ...

ATF_TP_ADD_TCS(tp, tcs)
{
   ATF_TP_ADD_TC(tcs, tc1)
   ATF_TP_ADD_TC(tcs, tc2)
   ... add additional test cases ...

   return atf_no_error();
}
```

Definition of test cases

Test cases have an identifier and are composed of three different parts: the header, the body and an optional cleanup routine, all of which are described in `atf-test-case`(8). To define test cases, one can use the **`ATF_TC`**() or the **`ATF_TC_WITH_CLEANUP`**() macros, which take a single parameter specifiying the test case's name. The former does not allow the specification of a cleanup routine for the test case while the latter does. It is important to note that these *do not* set the test case up for execution when the program is run. In order to do so, a later registration is needed with the **`ATF_TP_ADD_TC`**() macro detailed in **Program initialization**.

Later on, one must define the three parts of the body by means of three functions. Their headers are given by the **`ATF_TC_HEAD`**(), **`ATF_TC_BODY`**() and **`ATF_TC_CLEANUP`**() macros, all of which take the test case name provided to the **`ATF_TC`**() or **`ATF_TC_WITH_CLEANUP`**() macros. Following each of these, a block of code is expected, surrounded by the opening and closing brackets.

Program initialization

The library provides a way to easily define the test program's **`main`**() function. You should never define one on your own, but rely on the library to do it for you. This is done by using the **`ATF_TP_ADD_TCS`**() macro, which is passed the name of the object that will hold the test cases; i.e. the test program instance. This name can be whatever you want as long as it is a valid variable identifier.

After the macro, you are supposed to provide the body of a function, which should only use the **`ATF_TP_ADD_TC`**() macro to register the test cases the test program will execute and return a success error code. The first parameter of this macro matches the name you provided in the former call. The success status can be returned using the **`atf_no_error`**() function.

Header definitions
The test case's header can define the meta-data by using the **atf_tc_set_md_var**() method, which takes two parameters: the first one specifies the meta-data variable to be set and the second one specifies its value. Both of them are strings.

Configuration variables
The test case has read-only access to the current configuration variables by means of the *bool* **atf_tc_has_config_var**() and the *const char ** **atf_tc_get_config_var**() methods, which can be called in any of the three parts of a test case.

Access to the source directory
It is possible to get the path to the test case's source directory from any of its three components by querying the 'srcdir' configuration variable.

Requiring programs
Aside from the *require.progs* meta-data variable available in the header only, one can also check for additional programs in the test case's body by using the **atf_tc_require_prog**() function, which takes the base name or full path of a single binary. Relative paths are forbidden. If it is not found, the test case will be automatically skipped.

Test case finalization
The test case finalizes either when the body reaches its end, at which point the test is assumed to have *passed*, or at any explicit call to **atf_tc_pass**(), **atf_tc_fail**() or **atf_tc_skip**(). These three functions terminate the execution of the test case immediately. The cleanup routine will be processed afterwards in a completely automated way, regardless of the test case's termination reason.

atf_tc_pass() does not take any parameters. **atf_tc_fail**() and **atf_tc_skip**() take a single string that describes why the test case failed or was skipped, respectively. It is very important to provide a clear error message in both cases so that the user can quickly know why the test did not pass.

Helper macros for common checks
The library provides several macros that are very handy in multiple situations. These basically check some condition after executing a given statement or processing a given expression and, if the condition is not met, they automatically call **atf_tc_fail**() with an appropriate error message.

ATF_CHECK() takes an expression and raises a failure if it evaluates to false.

ATF_CHECK_EQUAL() takes two expressions and raises a failure if the two do not evaluate to the same exact value.

EXAMPLES
The following shows a complete test program with a single test case that validates the addition operator:

```
#include <atf-c.h>

ATF_TC(addition);
ATF_TC_HEAD(addition)
{
    atf_tc_set_md_var("descr", "Sample tests for the addition operator");
}
ATF_TC_BODY(addition)
{
    ATF_CHECK_EQUAL(0 + 0, 0);
    ATF_CHECK_EQUAL(0 + 1, 1);
```

```
    ATF_CHECK_EQUAL(1 + 0, 1);

    ATF_CHECK_EQUAL(1 + 1, 2);

    ATF_CHECK_EQUAL(100 + 200, 300);
}

ATF_TP_ADD_TCS(tp)
{
    ATF_TP_ADD_TC(tp, addition);

    return atf_no_error();
}
```

SEE ALSO

atf-test-program(1), atf(7), atf-test-case(8)

NAME

atf_add_test_case, **atf_check**, **atf_check_equal**, **atf_config_get**, **atf_config_has**, **atf_fail**, **atf_get**, **atf_get_srcdir**, **atf_pass**, **atf_require_prog**, **atf_set**, **atf_skip** — POSIX shell API to write ATF-based test programs

SYNOPSIS

atf_add_test_case(*name*);

atf_check(*command*);

atf_check_equal(*expr1*, *expr2*);

atf_config_get(*var_name*);

atf_config_has(*var_name*);

atf_fail(*reason*);

atf_get(*var_name*);

atf_get_srcdir();

atf_pass();

atf_require_prog(*prog_name*);

atf_set(*var_name*, *value*);

atf_skip(*reason*);

DESCRIPTION

ATF provides a simple but powerful interface to easily write test programs in the POSIX shell language. These are extremely helpful given that they are trivial to write due to the language simplicity and the great deal of available external tools, so they are often ideal to test other applications at the user level.

Test programs written using this library must be preprocessed by the `atf-compile`(1) tool, which includes some boilerplate code and generates the final (installable) test program.

Shell-based test programs always follow this template:

```
atf_test_case tc1
tc1_head() {
    ... first test case's header ...
}
tc1_body() {
    ... first test case's body ...
}

atf_test_case tc2
tc2_head() {
    ... second test case's header ...
}
tc2_body() {
    ... second test case's body ...
}
tc2_cleanup() {
    ... second test case's cleanup ...
}
```

```
. . . additional test cases . . .

atf_init_test_cases() {
   atf_add_test_case tc1
   atf_add_test_case tc2
   ... add additional test cases ...
}
```

Definition of test cases

Test cases have an identifier and are composed of three different parts: the header, the body and an optional cleanup routine, all of which are described in `atf-test-case`(8). To define test cases, one can use the **`atf_test_case`**() function, which takes a single parameter specifiying the test case's name and instructs the library to set things up to accept it as a valid test case. It is important to note that it *does not* set the test case up for execution when the program is run. In order to do so, a later registration is needed through the **`atf_add_test_case`**() function detailed in **Program initialization**.

Later on, one must define the three parts of the body by providing two or three functions (remember that the cleanup routine is optional). These functions are named after the test case's identifier, and are **`<id>_head`**(), **`<id>_body`**() and **`<id>_cleanup.`**() None of these take parameters when executed.

Program initialization

The test program must define an **`atf_init_test_cases`**() function, which is in charge of registering the test cases that will be executed at run time by using the **`atf_add_test_case`**() function, which takes the name of a test case as its single parameter. This main function should not do anything else, except maybe sourcing auxiliary source files that define extra variables and functions.

Configuration variables

The test case has read-only access to the current configuration variables through the **`atf_config_has`**() and **`atf_config_get`**() methods. The former takes a single parameter specifying a variable name and returns a boolean indicating whether the variable is defined or not. The latter can take one or two parameters. If it takes only one, it specifies the variable from which to get the value, and this variable must be defined. If it takes two, the second one specifies a default value to be returned if the variable is not available.

Access to the source directory

It is possible to get the path to the test case's source directory from anywhere in the test program by using the **`atf_get_srcdir`**() function. It is interesting to note that this can be used inside **`atf_init_test_cases`**() to silently include additional helper files from the source directory.

Requiring programs

Aside from the *require.progs* meta-data variable available in the header only, one can also check for additional programs in the test case's body by using the **`atf_require_prog`**() function, which takes the base name or full path of a single binary. Relative paths are forbidden. If it is not found, the test case will be automatically skipped.

Test case finalization

The test case finalizes either when the body reaches its end, at which point the test is assumed to have *passed*, or at any explicit call to **`atf_pass`**(), **`atf_fail`**() or **`atf_skip`**(). These three functions terminate the execution of the test case immediately. The cleanup routine will be processed afterwards in a completely automated way, regardless of the test case's termination reason.

`atf_pass`() does not take any parameters. **`atf_fail`**() and **`atf_skip`**() take a single string parameter that describes why the test case failed or was skipped, respectively. It is very important to provide a clear error message in both cases so that the user can quickly know why the test did not pass.

Helper functions for common checks

atf_check(*cmd*, *expcode*, *expout*, *experr*)

This function takes four parameters: the command to execute, the expected numerical exit code, the expected behavior of `stdout` and the expected behavior of `stderr`. *expout* can be one of the following:

`expout` What the command writes to the `stdout` channel must match exactly what is found in the `expout` file.

`ignore` The test does not check what the command writes to the `stdout` channel.

`null` The command must not write anything to the `stdout` channel.

`stdout` What the command writes to the `stdout` channel is written to a `stdout` file, available for further inspection.

Similarly, *experr* can be one of '`experr`', '`ignore`', '`null`', or '`stderr`', all of which follow the same semantics of their corresponding counterparts for the *expout* case.

It is important to note that when a failure is detected, this function will print as much information as possible to be able to identify the cause of the failure. For example, if the `stdout` does not match with the expected contents, a diff will be printed.

atf_check_equal(*expr1*, *expr2*)

This function takes two expressions, evaluates them and, if their results differ, aborts the test case with an appropriate failure message.

EXAMPLES

The following shows a complete test program with a single test case that validates the addition operator:

```
atf_test_case addition
addition_head() {
    atf_set "descr" "Sample tests for the addition operator"
}
addition_body() {
    atf_check_equal $((0 + 0)) 0
    atf_check_equal $((0 + 1)) 1
    atf_check_equal $((1 + 0)) 0

    atf_check_equal $((1 + 1)) 2

    atf_check_equal $((100 + 200)) 300
}

atf_init_test_cases() {
    atf_add_test_case addition
}
```

This other example shows how to include a file with extra helper functions in the test program:

```
. . . definition of test cases . . .

atf_init_test_cases() {
   . $(atf_get_srcdir)/helper_functions.sh

   atf_add_test_case foo1
   atf_add_test_case foo2
```

```
}
```

This example demonstrates the use of the very useful **atf_check**() function:

```
# Check for silent output
atf_check 'true' 0 null null

# Check for silent output and failure
atf_check 'false' 1 null null

# Check for known stdout and silent stderr
echo foo >expout
atf_check 'echo foo' 0 expout null

# Generate a file for later inspection
atf_check 'ls' 0 stdout null
grep foo ls || atf_fail "foo file not found in listing"
```

SEE ALSO

atf-compile(1), atf-test-program(1), atf(7), atf-test-case(8)

NAME

atof — convert ASCII string to double

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

double
atof(const char *nptr);
```

DESCRIPTION

The **atof**() function converts the initial portion of the string pointed to by *nptr* to *double* representation.

It is equivalent to:

```
strtod(nptr, (char **)NULL);
```

SEE ALSO

atoi(3), atol(3), strtod(3), strtol(3), strtoul(3)

STANDARDS

The **atof**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

atoi — convert ASCII string to integer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
atoi(const char *nptr);
```

DESCRIPTION

The **atoi**() function converts the initial portion of the string pointed to by *nptr* to *integer* representation.

It is equivalent to:

```
(int)strtol(nptr, (char **)NULL, 10);
```

SEE ALSO

atof(3), atol(3), strtod(3), strtol(3), strtoul(3)

STANDARDS

The **atoi**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

CAVEATS

atoi does no overflow checking, handles unsigned numbers poorly, and handles strings containing trailing extra characters (like “123abc”) poorly. Careful use of strtol(3) and strtoul(3) can alleviate these problems.

NAME

atol — convert ASCII string to long integer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

long
atol(const char *nptr);
```

DESCRIPTION

The **atol**() function converts the initial portion of the string pointed to by *nptr* to *long integer* representation.

It is equivalent to:

```
strtol(nptr, (char **)NULL, 10);
```

SEE ALSO

atof(3), atoi(3), strtod(3), strtol(3), strtoul(3)

STANDARDS

The **atol**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

atoll — convert ASCII string to long long integer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

long long int
atoll(const char *nptr);
```

DESCRIPTION

The **atoll**() function converts the initial portion of the string pointed to by *nptr* to *long long integer* representation.

It is equivalent to:

```
strtoll(nptr, (char **)NULL, 10);
```

SEE ALSO

atof(3), atoi(3), atol(3), strtod(3), strtol(3), strtoll(3), strtoul(3), strtoull(3)

STANDARDS

The **atoll**() function conforms to ISO/IEC 9899:1999 (“ISO C99”).

NAME

atomic_add, **atomic_add_32**, **atomic_add_int**, **atomic_add_long**, **atomic_add_ptr**, **atomic_add_64**, **atomic_add_32_nv**, **atomic_add_int_nv**, **atomic_add_long_nv**, **atomic_add_ptr_nv**, **atomic_add_64_nv** — atomic add operations

SYNOPSIS

```
#include <sys/atomic.h>

void
atomic_add_32(volatile uint32_t *ptr, int32_t delta);

void
atomic_add_int(volatile unsigned int *ptr, int delta);

void
atomic_add_long(volatile unsigned long *ptr, long delta);

void
atomic_add_ptr(volatile void *ptr, ssize_t delta);

void
atomic_add_64(volatile uint64_t *ptr, int64_t delta);

uint32_t
atomic_add_32_nv(volatile uint32_t *ptr, int32_t delta);

unsigned int
atomic_add_int_nv(volatile unsigned int *ptr, int delta);

unsigned long
atomic_add_long_nv(volatile unsigned long *ptr, long delta);

void *
atomic_add_ptr_nv(volatile void *ptr, ssize_t delta);

uint64_t
atomic_add_64_nv(volatile uint64_t *ptr, int64_t delta);
```

DESCRIPTION

The **atomic_add** family of functions add a signed value *delta* to the variable referenced by *ptr* in an atomic fashion.

The *_**nv**() variants of these functions return the new value.

The 64-bit variants of these functions are available only on platforms that can support atomic 64-bit memory access. Applications can check for the availability of 64-bit atomic memory operations by testing if the pre-processor macro __HAVE_ATOMIC64_OPS is defined.

SEE ALSO

atomic_ops(3)

HISTORY

The **atomic_add** functions first appeared in NetBSD 5.0.

NAME

atomic_and, **atomic_and_32**, **atomic_and_uint**, **atomic_and_ulong**, **atomic_and_64**, **atomic_and_32_nv**, **atomic_and_uint_nv**, **atomic_and_ulong_nv**, **atomic_and_64_nv** — atomic logical ‘and’ operations

SYNOPSIS

```
#include <sys/atomic.h>

void
atomic_and_32(volatile uint32_t *ptr, uint32_t bits);

void
atomic_and_uint(volatile unsigned int *ptr, unsigned int bits);

void
atomic_and_ulong(volatile unsigned long *ptr, unsigned long bits);

void
atomic_and_64(volatile uint64_t *ptr, uint64_t bits);

uint32_t
atomic_and_32_nv(volatile uint32_t *ptr, uint32_t bits);

unsigned int
atomic_and_uint_nv(volatile unsigned int *ptr, unsigned int bits);

unsigned long
atomic_and_ulong_nv(volatile unsigned long *ptr, unsigned long bits);

uint64_t
atomic_and_64_nv(volatile uint64_t *ptr, uint64_t bits);
```

DESCRIPTION

The **atomic_and** family of functions load the value of the variable referenced by *ptr*, perform a logical ‘and’ with the value *bits*, and store the result back to the variable referenced by *ptr* in an atomic fashion.

The *_**nv**() variants of these functions return the new value.

The 64-bit variants of these functions are available only on platforms that can support atomic 64-bit memory access. Applications can check for the availability of 64-bit atomic memory operations by testing if the preprocessor macro __HAVE_ATOMIC64_OPS is defined.

SEE ALSO

atomic_ops(3)

HISTORY

The **atomic_and** functions first appeared in NetBSD 5.0.

NAME

atomic_cas, **atomic_cas_32**, **atomic_cas_uint**, **atomic_cas_ulong**, **atomic_cas_ptr**, **atomic_cas_64** — atomic compare-and-swap operations

SYNOPSIS

```
#include <sys/atomic.h>

uint32_t
atomic_cas_32(volatile uint32_t *ptr, uint32_t old, uint32_t new);

unsigned int
atomic_cas_uint(volatile unsigned int *ptr, unsigned int old,
    unsigned int new);

unsigned long
atomic_cas_ulong(volatile unsigned long *ptr, unsigned long old,
    unsigned long new);

void *
atomic_cas_ptr(volatile void *ptr, void *old, void *new);

uint64_t
atomic_cas_64(volatile uint64_t *ptr, uint64_t old, uint64_t new);
```

DESCRIPTION

The **atomic_cas** family of functions perform a compare-and-swap operation in an atomic fashion. The value of the variable referenced by *ptr* is compared against *old*. If the values are equal, *new* is stored in the variable referenced by *ptr*.

The old value of the variable referenced by *ptr* is always returned regardless of whether or not the new value was stored. Applications can test for success of the operation by comparing the return value to the value passed as *old*; if they are equal then the new value was stored.

The 64-bit variants of these functions are available only on platforms that can support atomic 64-bit memory access. Applications can check for the availability of 64-bit atomic memory operations by testing if the pre-processor macro `__HAVE_ATOMIC64_OPS` is defined.

SEE ALSO

atomic_ops(3)

HISTORY

The **atomic_cas** functions first appeared in NetBSD 5.0.

NAME

atomic_dec, **atomic_dec_32**, **atomic_dec_uint**, **atomic_dec_ulong**, **atomic_dec_ptr**, **atomic_dec_64**, **atomic_dec_32_nv**, **atomic_dec_uint_nv**, **atomic_dec_ulong_nv**, **atomic_dec_ptr_nv**, **atomic_dec_64_nv** — atomic decrement operations

SYNOPSIS

```
#include <sys/atomic.h>

void
atomic_dec_32(volatile uint32_t *ptr);

void
atomic_dec_uint(volatile unsigned int *ptr);

void
atomic_dec_ulong(volatile unsigned long *ptr);

void
atomic_dec_ptr(volatile void *ptr);

void
atomic_dec_64(volatile uint64_t *ptr);

uint32_t
atomic_dec_32_nv(volatile uint32_t *ptr);

unsigned int
atomic_dec_uint_nv(volatile unsigned int *ptr);

unsigned long
atomic_dec_ulong_nv(volatile unsigned long *ptr);

void *
atomic_dec_ptr_nv(volatile void *ptr);

uint64_t
atomic_dec_64_nv(volatile uint64_t *ptr);
```

DESCRIPTION

The **atomic_dec** family of functions decrement (by one) the variable referenced by *ptr* in an atomic fashion.

The *_**nv**() variants of these functions return the new value.

The 64-bit variants of these functions are available only on platforms that can support atomic 64-bit memory access. Applications can check for the availability of 64-bit atomic memory operations by testing if the pre-processor macro __HAVE_ATOMIC64_OPS is defined.

SEE ALSO

atomic_ops(3)

HISTORY

The **atomic_dec** functions first appeared in NetBSD 5.0.

NAME

atomic_inc, **atomic_inc_32**, **atomic_inc_uint**, **atomic_inc_ulong**, **atomic_inc_ptr**, **atomic_inc_64**, **atomic_inc_32_nv**, **atomic_inc_uint_nv**, **atomic_inc_ulong_nv**, **atomic_inc_ptr_nv**, **atomic_inc_64_nv** — atomic increment operations

SYNOPSIS

```
#include <sys/atomic.h>

void
atomic_inc_32(volatile uint32_t *ptr);

void
atomic_inc_uint(volatile unsigned int *ptr);

void
atomic_inc_ulong(volatile unsigned long *ptr);

void
atomic_inc_ptr(volatile void *ptr);

void
atomic_inc_64(volatile uint64_t *ptr);

uint32_t
atomic_inc_32_nv(volatile uint32_t *ptr);

unsigned int
atomic_inc_uint_nv(volatile unsigned int *ptr);

unsigned long
atomic_inc_ulong_nv(volatile unsigned long *ptr);

void *
atomic_inc_ptr_nv(volatile void *ptr);

uint64_t
atomic_inc_64_nv(volatile uint64_t *ptr);
```

DESCRIPTION

The **atomic_inc** family of functions increment (by one) the variable referenced by *ptr* in an atomic fashion.

The *_**nv**() variants of these functions return the new value.

The 64-bit variants of these functions are available only on platforms that can support atomic 64-bit memory access. Applications can check for the availability of 64-bit atomic memory operations by testing if the pre-processor macro __HAVE_ATOMIC64_OPS is defined.

SEE ALSO

atomic_ops(3)

HISTORY

The **atomic_inc** functions first appeared in NetBSD 5.0.

NAME

atomic_ops — atomic memory operations

SYNOPSIS

```
#include <sys/atomic.h>
```

DESCRIPTION

The **atomic_ops** family of functions provide atomic memory operations. There are 7 classes of atomic memory operations available :

`atomic_add`(3) These functions perform atomic addition.

`atomic_and`(3) These functions perform atomic logical “and”.

`atomic_cas`(3) These functions perform atomic compare-and-swap.

`atomic_dec`(3) These functions perform atomic decrement.

`atomic_inc`(3) These functions perform atomic increment.

`atomic_or`(3) These functions perform atomic logical “or”.

`atomic_swap`(3) These functions perform atomic swap.

Synchronization mechanisms

Where the architecture does not provide hardware support for atomic compare and swap (CAS), atomicity is provided by a restartable sequence or by a spinlock. The chosen method is not ordinarily distinguishable by or visible to users of the interface. The following architectures can be assumed to provide CAS in hardware: alpha, amd64, i386, powerpc, powerpc64, sparc64.

Scope and restrictions

If hardware CAS is available, the atomic operations are globally atomic: operations within a memory region shared between processes are guaranteed to be performed atomically. If hardware CAS is not available, it may only be assumed that the operations are atomic with respect to threads in the same process. Additionally, if hardware CAS is not available, the atomic operations must not be used within a signal handler.

Users of atomic memory operations should not make assumptions about how the memory access is performed (specifically, the width of the memory access). For this reason, applications making use of atomic memory operations should limit their use to regular memory. The results of using atomic memory operations on anything other than regular memory are undefined.

Users of atomic memory operations should take care to modify any given memory location either entirely with atomic operations or entirely with some other synchronization mechanism. Intermixing of atomic operations with other synchronization mechanisms for the same memory location results in undefined behavior.

Visibility and ordering of memory accesses

If hardware CAS is available, stores to the target memory location by an atomic operation will reach global visibility before the operation completes. If hardware CAS is not available, the store may not reach global visibility until some time after the atomic operation has completed. However, in all cases a subsequent atomic operation on the same memory cell will be delayed until the result of any preceeding operation has reached global visibility.

Atomic operations are strongly ordered with respect to each other. The global visibility of other loads and stores before and after an atomic operation is undefined. Applications that require synchronization of loads and stores with respect to an atomic operation must use memory barriers. See

`membar_ops`(3).

Performance

Because atomic memory operations require expensive synchronization at the hardware level, applications should take care to minimize their use. In certain cases, it may be more appropriate to use a mutex, especially if more than one memory location will be modified.

SEE ALSO

`atomic_add`(3), `atomic_and`(3), `atomic_cas`(3), `atomic_dec`(3), `atomic_inc`(3), `atomic_or`(3), `atomic_swap`(3), `membar_ops`(3)

HISTORY

The **`atomic_ops`** functions first appeared in NetBSD 5.0.

NAME

atomic_or, **atomic_or_32**, **atomic_or_uint**, **atomic_or_ulong**, **atomic_or_64**, **atomic_or_32_nv**, **atomic_or_uint_nv**, **atomic_or_ulong_nv**, **atomic_or_64_nv** — atomic logical ‘or’ operations

SYNOPSIS

```
#include <sys/atomic.h>

void
atomic_or_32(volatile uint32_t *ptr, uint32_t bits);

void
atomic_or_uint(volatile unsigned int *ptr, unsigned int bits);

void
atomic_or_ulong(volatile unsigned long *ptr, unsigned long bits);

void
atomic_or_64(volatile uint64_t *ptr, uint64_t bits);

uint32_t
atomic_or_32_nv(volatile uint32_t *ptr, uint32_t bits);

unsigned int
atomic_or_uint_nv(volatile unsigned int *ptr, unsigned int bits);

unsigned long
atomic_or_ulong_nv(volatile unsigned long *ptr, unsigned long bits);

uint64_t
atomic_or_64_nv(volatile uint64_t *ptr, uint64_t bits);
```

DESCRIPTION

The **atomic_or** family of functions load the value of the variable referenced by *ptr*, perform a logical ‘or’ with the value *bits*, and store the result back to the variable referenced by *ptr* in an atomic fashion.

The *_**nv**() variants of these functions return the new value.

The 64-bit variants of these functions are available only on platforms that can support atomic 64-bit memory access. Applications can check for the availability of 64-bit atomic memory operations by testing if the preprocessor macro __HAVE_ATOMIC64_OPS is defined.

SEE ALSO

atomic_ops(3)

HISTORY

The **atomic_or** functions first appeared in NetBSD 5.0.

NAME

atomic_swap, **atomic_swap_32**, **atomic_swap_uint**, **atomic_swap_ulong**, **atomic_swap_ptr**, **atomic_swap_64** — atomic swap operations

SYNOPSIS

```
#include <sys/atomic.h>

uint32_t
atomic_swap_32(volatile uint32_t *ptr, uint32_t new);

unsigned int
atomic_swap_uint(volatile unsigned int *ptr, unsigned int new);

unsigned long
atomic_swap_ulong(volatile unsigned long *ptr, unsigned long new);

void *
atomic_swap_ptr(volatile void *ptr, void *new);

uint64_t
atomic_swap_64(volatile uint64_t *ptr, uint64_t new);
```

DESCRIPTION

The **atomic_swap** family of functions perform a swap operation in an atomic fashion. The value of the variable referenced by *ptr* is replaced by *new* and the old value returned.

The 64-bit variants of these functions are available only on platforms that can support atomic 64-bit memory access. Applications can check for the availability of 64-bit atomic memory operations by testing if the preprocessor macro __HAVE_ATOMIC64_OPS is defined.

SEE ALSO

atomic_ops(3)

HISTORY

The **atomic_swap** functions first appeared in NetBSD 5.0.

NAME

basename — return the last component of a pathname

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <libgen.h>

char *
basename(char *path);
```

DESCRIPTION

The **basename**() function takes the pathname pointed to by *path* and returns a pointer to the final component of the pathname, deleting any trailing '/' characters.

If *path* consists entirely of '/' characters, **basename**() returns a pointer to the string "/".

If *path* is a null pointer or points to an empty string, **basename**() returns a pointer to the string ".".

RETURN VALUES

The **basename**() function returns a pointer to the final component of *path*.

SEE ALSO

basename(1), dirname(3)

STANDARDS

- X/Open Portability Guide Issue 4, Version 2 ("XPG4.2")
- IEEE Std 1003.1-2001 ("POSIX.1")

BUGS

If the length of the result is longer than PATH_MAX bytes (including the terminating nul), the result will be truncated.

The **basename**() function returns a pointer to static storage that may be overwritten by subsequent calls to **basename**(). This is not strictly a bug; it is explicitly allowed by IEEE Std 1003.1-2001 ("POSIX.1").

NAME

bcmp — compare byte string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <strings.h>

int
bcmp(const void *b1, const void *b2, size_t len);
```

DESCRIPTION

The **bcmp**() function compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *len* bytes long. Zero-length strings are always identical.

The strings may overlap.

SEE ALSO

memcmp(3), strcasecmp(3), strcmp(3), strcoll(3), strxfrm(3)

HISTORY

A **bcmp**() function first appeared in 4.2BSD.

NAME

bcopy — copy byte string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <strings.h>

void
bcopy(const void *src, void *dst, size_t len);
```

DESCRIPTION

The **bcopy**() function copies *len* bytes from string *src* to string *dst*. The two strings may overlap. If *len* is zero, no bytes are copied.

SEE ALSO

memccpy(3), memcpy(3), memmove(3), strcpy(3), strncpy(3)

HISTORY

A **bcopy**() function appeared in 4.2BSD.

NAME

bind_textdomain_codeset − set encoding of message translations

SYNOPSIS

#include <libintl.h>

char * bind_textdomain_codeset (const char * *domainname***,**
const char * *codeset***);**

DESCRIPTION

The **bind_textdomain_codeset** function sets the output codeset for message catalogs for domain *domainname*.

A message domain is a set of translatable *msgid* messages. Usually, every software package has its own message domain.

By default, the **gettext** family of functions returns translated messages in the locale's character encoding, which can be retrieved as **nl_langinfo(CODESET)**. The need for calling **bind_textdomain_codeset** arises for programs which store strings in a locale independent way (e.g. UTF-8) and want to avoid an extra character set conversion on the returned translated messages.

domainname must be a non-empty string.

If *codeset* is not NULL, it must be a valid encoding name which can be used for the **iconv_open** function. The **bind_textdomain_codeset** function sets the output codeset for message catalogs belonging to domain *domainname* to *codeset*. The function makes copies of the argument strings as needed.

If *codeset* is NULL, the function returns the previously set codeset for domain *domainname*. The default is NULL, denoting the locale's character encoding.

RETURN VALUE

If successful, the **bind_textdomain_codeset** function returns the current codeset for domain *domainname*, after possibly changing it. The resulting string is valid until the next **bind_textdomain_codeset** call for the same *domainname* and must not be modified or freed. If a memory allocation failure occurs, it sets **errno** to **ENOMEM** and returns NULL. If no codeset has been set for domain *domainname*, it returns NULL.

ERRORS

The following error can occur, among others:

ENOMEM
Not enough memory available.

BUGS

The return type ought to be **const char ***, but is **char *** to avoid warnings in C code predating ANSI C.

SEE ALSO

gettext(3), **dgettext**(3), **dcgettext**(3), **ngettext**(3), **dngettext**(3), **dcngettext**(3), **textdomain**(3), **nl_langinfo**(3), **iconv_open**(3)

NAME

bindresvport, **bindresvport_sa** — bind a socket to a reserved privileged IP port

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <rpc/rpc.h>

int
bindresvport(int sd, struct sockaddr_in *sin);

int
bindresvport_sa(int sd, struct sockaddr *sa);
```

DESCRIPTION

bindresvport() and **bindresvport_sa**() are used to bind a socket descriptor to a reserved privileged IP port, that is, a port number in the range 0-1023. The routine returns 0 if it is successful, otherwise -1 is returned and *errno* set to reflect the cause of the error.

If *sin* is a pointer to a *struct sockaddr_in* then the appropriate fields in the structure should be defined. Note that *sin->sin_family* must be initialized to the address family of the socket, passed by *sd*. If *sin->sin_port* is '0' then a port (in the range 600-1023) will be chosen, and if bind(2) is successful, the *sin->sin_port* will be updated to contain the allocated port.

If *sin* is the NULL pointer, a port will be allocated (as above). However, there is no way for **bindresvport**() to return the allocated port in this case. getsockname(2) can be used to determine the assigned port.

Only root can bind to a privileged port; this call will fail for any other users.

Function prototype of **bindresvport**() is biased to AF_INET socket. **bindresvport_sa**() acts exactly the same, with more neutral function prototype. Note that both functions behave exactly the same, and both support AF_INET6 sockets as well as AF_INET sockets.

RETURN VALUES

If the bind is successful, a 0 value is returned. A return value of -1 indicates an error, which is further specified in the global *errno*.

ERRORS

[EPFNOSUPPORT] If second argument was supplied, and address family did not match between arguments.

bindresvport() may also fail and set *errno* for any of the errors specified for the calls bind(2), getsockopt(2), or setsockopt(2).

SEE ALSO

bind(2), getsockname(2), getsockopt(2), setsockopt(2), ip(4)

NAME

bindtextdomain − set directory containing message catalogs

SYNOPSIS

#include <libintl.h>

char * bindtextdomain (const char * *domainname***, const char *** *dirname***);**

DESCRIPTION

The **bindtextdomain** function sets the base directory of the hierarchy containing message catalogs for a given message domain.

A message domain is a set of translatable *msgid* messages. Usually, every software package has its own message domain. The need for calling **bindtextdomain** arises because packages are not always installed with the same prefix as the <libintl.h> header and the libc/libintl libraries.

Message catalogs will be expected at the pathnames *dirname*/*locale*/*category*/*domainname*.mo, where *locale* is a locale name and *category* is a locale facet such as **LC_MESSAGES**.

domainname must be a non-empty string.

If *dirname* is not NULL, the base directory for message catalogs belonging to domain *domainname* is set to *dirname*. The function makes copies of the argument strings as needed. If the program wishes to call the **chdir** function, it is important that *dirname* be an absolute pathname; otherwise it cannot be guaranteed that the message catalogs will be found.

If *dirname* is NULL, the function returns the previously set base directory for domain *domainname*.

RETURN VALUE

If successful, the **bindtextdomain** function returns the current base directory for domain *domainname*, after possibly changing it. The resulting string is valid until the next **bindtextdomain** call for the same *domainname* and must not be modified or freed. If a memory allocation failure occurs, it sets **errno** to **ENOMEM** and returns NULL.

ERRORS

The following error can occur, among others:

ENOMEM

Not enough memory available.

BUGS

The return type ought to be **const char ***, but is **char *** to avoid warnings in C code predating ANSI C.

SEE ALSO

gettext(3), **dgettext**(3), **dcgettext**(3), **ngettext**(3), **dngettext**(3), **dcngettext**(3), **textdomain**(3), **realpath**(3)

NAME

__BIT, **__BITS**, **__SHIFTIN**, **__SHIFTOUT**, **__SHIFTOUT_MASK** — macros for preparing bitmasks and operating on bit fields

SYNOPSIS

```
#include <sys/cdefs.h>

uint32_t
__BIT(n);

uint32_t
__BITS(m, n);

__SHIFTIN(v, mask);

__SHIFTOUT(v, mask);

__SHIFTOUT_MASK(mask);
```

DESCRIPTION

These macros prepare bitmasks, extract bitfields from words, and insert bitfields into words. A "bitfield" is a span of consecutive bits defined by a bitmask, where 1s select the bits in the bitfield.

Use __BIT and __BITS to define bitmasks:

__BIT(*n*)
: Return a bitmask with bit m set, where the least significant bit is bit 0.

__BITS(*m*, *n*)
: Return a bitmask with bits *m* through *n*, inclusive, set. It does not matter whether *m* > *n* or *m* <= *n*. The least significant bit is bit 0.

__SHIFTIN(), **__SHIFTOUT**(), and **__SHIFTOUT_MASK**() help read and write bitfields from words:

__SHIFTIN(*v*, *mask*)
: Left-shift bits *v* into the bitfield defined by *mask*, and return them. No side-effects.

__SHIFTOUT(*v*, *mask*)
: Extract and return the bitfield selected by *mask* from *v*, right-shifting the bits so that the rightmost selected bit is at bit 0. No side-effects.

__SHIFTOUT_MASK(*mask*)
: Right-shift the bits in *mask* so that the rightmost non-zero bit is at bit 0. This is useful for finding the greatest unsigned value that a bitfield can hold. No side-effects. Note that **__SHIFTOUT_MASK**(*m*, =) **__SHIFTOUT**(*m*, *m*).

EXAMPLES

```
/*
 * Register definitions taken from the RFMD RF3000 manual.
 */
#define RF3000_GAINCTL          0x11            /* TX variable gain control */
#define         RF3000_GAINCTL_TXVGC_MASK       __BITS(7, 2)
#define         RF3000_GAINCTL_SCRAMBLER        __BIT(1)

/*
 * Shift the transmit power into the transmit-power field of the
 * gain-control register and write it to the baseband processor.
```

```
 */
atw_rf3000_write(sc, RF3000_GAINCTL,
    __SHIFTIN(txpower, RF3000_GAINCTL_TXVGC_MASK));

/*
 * Register definitions taken from the ADMtek ADM8211 manual.
 *
 */
#define ATW_RXSTAT_OWN __BIT(31)             /* 1: NIC may fill descriptor */
/* ... */
#define ATW_RXSTAT_DA1 __BIT(17)             /* DA bit 1, admin'd address */
#define ATW_RXSTAT_DA0          __BIT(16)    /* DA bit 0, group address */
#define ATW_RXSTAT_RXDR_MASK    __BITS(15,12) /* RX data rate */
#define ATW_RXSTAT_FL_MASK     __BITS(11,0)  /* RX frame length, last
                                              * descriptor only
                                              */

/* Extract the frame length from the Rx descriptor's
 * status field.
 */
len = __SHIFTOUT(rxstat, ATW_RXSTAT_FL_MASK);
```

HISTORY

The **bits** macros first appeared in atw(4), with different names and implementation. **bits** macros appeared with their current names and implementation in NetBSD 4.0.

AUTHORS

The **bits** macros were written by David Young ⟨dyoung@NetBSD.org⟩.
Matt Thomas ⟨matt@NetBSD.org⟩ suggested important improvements to the implementation, and contributed the macro names **SHIFTIN**() and **SHIFTOUT**().

BUGS

__BIT() and **__BITS**() can only express 32-bit bitmasks.

NAME

bit_alloc, **bit_clear**, **bit_decl**, **bit_ffc**, **bit_ffs**, **bit_nclear**, **bit_nset**, **bit_set**, **bitstr_size**, **bit_test** — bit-string manipulation macros

SYNOPSIS

```
#include <bitstring.h>

bitstr_t *
bit_alloc(int nbits);

bit_clear(bit_str name, int bit);

bit_decl(bit_str name, int nbits);

bit_ffc(bit_str name, int nbits, int *value);

bit_ffs(bit_str name, int nbits, int *value);

bit_nclear(bit_str name, int start, int stop);

bit_nset(bit_str name, int start, int stop);

bit_set(bit_str name, int bit);

bitstr_size(int nbits);

bit_test(bit_str name, int bit);
```

DESCRIPTION

These macros operate on strings of bits.

The macro **bit_alloc**() returns a pointer of type "*bitstr_t **" to sufficient space to store *nbits* bits, or NULL if no space is available.

The macro **bit_decl**() allocates sufficient space to store *nbits* bits on the stack.

The macro **bitstr_size**() returns the number of elements of type *bitstr_t* necessary to store *nbits* bits. This is useful for copying bit strings.

The macros **bit_clear**() and **bit_set**() clear or set the zero-based numbered bit *bit*, in the bit string *name*.

The **bit_nset**() and **bit_nclear**() macros set or clear the zero-based numbered bits from *start* to *stop* in the bit string *name*.

The **bit_test**() macro evaluates to non-zero if the zero-based numbered bit *bit* of bit string *name* is set, and zero otherwise.

The **bit_ffs**() macro stores in the location referenced by *value* the zero-based number of the first bit set in the array of *nbits* bits referenced by *name*. If no bits are set, the location referenced by *value* is set to −1.

The macro **bit_ffc**() stores in the location referenced by *value* the zero-based number of the first bit not set in the array of *nbits* bits referenced by *name*. If all bits are set, the location referenced by *value* is set to −1.

The arguments to these macros are evaluated only once and may safely have side effects.

EXAMPLES

```
#include <limits.h>
#include <bitstring.h>
```

```
...
#define LPR_BUSY_BIT          0
#define LPR_FORMAT_BIT        1
#define LPR_DOWNLOAD_BIT      2
...
#define LPR_AVAILABLE_BIT     9
#define LPR_MAX_BITS          10

make_lpr_available()
{
        bitstr_t bit_decl(bitlist, LPR_MAX_BITS);
        ...
        bit_nclear(bitlist, 0, LPR_MAX_BITS - 1);
        ...
        if (!bit_test(bitlist, LPR_BUSY_BIT)) {
                bit_clear(bitlist, LPR_FORMAT_BIT);
                bit_clear(bitlist, LPR_DOWNLOAD_BIT);
                bit_set(bitlist, LPR_AVAILABLE_BIT);
        }
}
```

SEE ALSO

`malloc`(3)

HISTORY

The **`bitstring`** functions first appeared in 4.4BSD.

NAME

bt_gethostbyname, **bt_gethostbyaddr**, **bt_gethostent**, **bt_sethostent**, **bt_endhostent**, **bt_getprotobyname**, **bt_getprotobynumber**, **bt_getprotoent**, **bt_setprotoent**, **bt_endprotoent**, **bt_aton**, **bt_ntoa**, **bt_devaddr**, **bt_devname**, — Bluetooth routines

LIBRARY

library “libbluetooth”

SYNOPSIS

```
#include <bluetooth.h>

struct hostent *
bt_gethostbyname(const char *name);

struct hostent *
bt_gethostbyaddr(const char *addr, int len, int type);

struct hostent *
bt_gethostent(void);

void
bt_sethostent(int stayopen);

void
bt_endhostent(void);

struct protoent *
bt_getprotobyname(const char *name);

struct protoent *
bt_getprotobynumber(int proto);

struct protoent *
bt_getprotoent(void);

void
bt_setprotoent(int stayopen);

void
bt_endprotoent(void);

int
bt_aton(const char *str, bdaddr_t *ba);

const char *
bt_ntoa(const bdaddr_t *ba, char *str);

int
bt_devaddr(const char *name, bdaddr_t *addr);

int
bt_devname(char *name, const bdaddr_t *addr);
```

DESCRIPTION

The **bt_gethostent**(), **bt_gethostbyname**(), and **bt_gethostbyaddr**() functions each return a pointer to an object with the *hostent* structure describing a Bluetooth host referenced by name or by address, respectively.

The *name* argument passed to **bt_gethostbyname**() should point to a NUL-terminated hostname. The *addr* argument passed to **bt_gethostbyaddr**() should point to an address which is *len* bytes long, in binary form (i.e., not a Bluetooth BD_ADDR in human readable ASCII form). The *type* argument specifies the address family of this address and must be set to AF_BLUETOOTH.

The structure returned contains the information obtained from a line in /etc/bluetooth/hosts file.

The **bt_sethostent**() function controls whether /etc/bluetooth/hosts file should stay open after each call to **bt_gethostbyname**() or **bt_gethostbyaddr**(). If the *stayopen* flag is non-zero, the file will not be closed.

The **bt_endhostent**() function closes the /etc/bluetooth/hosts file.

The **bt_getprotoent**(), **bt_getprotobyname**(), and **bt_getprotobynumber**() functions each return a pointer to an object with the *protoent* structure describing a Bluetooth Protocol Service Multiplexor referenced by name or number, respectively.

The *name* argument passed to **bt_getprotobyname**() should point to a NUL-terminated Bluetooth Protocol Service Multiplexor name. The *proto* argument passed to **bt_getprotobynumber**() should have numeric value of the desired Bluetooth Protocol Service Multiplexor.

The structure returned contains the information obtained from a line in /etc/bluetooth/protocols file.

The **bt_setprotoent**() function controls whether /etc/bluetooth/protocols file should stay open after each call to **bt_getprotobyname**() or **bt_getprotobynumber**(). If the *stayopen* flag is non-zero, the file will not be closed.

The **bt_endprotoent**() function closes the /etc/bluetooth/protocols file.

The **bt_aton**() routine interprets the specified character string as a Bluetooth address, placing the address into the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid.

The routine **bt_ntoa**() takes a Bluetooth address and places an ASCII string representing the address into the buffer provided. It is up to the caller to ensure that provided buffer has enough space. If no buffer was provided then an internal static buffer will be used.

The **bt_devaddr**() function interprets the specified character string as the address or device name of a Bluetooth device on the local system, and places the device address in the structure provided, if any. It returns 1 if the string was successfully interpreted, or 0 if the string did not match any local device. The **bt_devname**() function takes a Bluetooth device address and copies the local device name associated with that address into the buffer provided, if any. It returns 1 when the device was found, otherwise 0.

FILES

/etc/bluetooth/hosts
/etc/bluetooth/protocols

EXAMPLES

Print out the hostname associated with a specific BD_ADDR:

```
const char *bdstr = "00:01:02:03:04:05";
bdaddr_t bd;
struct hostent *hp;

if (!bt_aton(bdstr, &bd))
        errx(1, "can't parse BD_ADDR %s", bdstr);

if ((hp = bt_gethostbyaddr((const char *)&bd,
```

```
        sizeof(bd), AF_BLUETOOTH)) == NULL)
            errx(1, "no name associated with %s", bdstr);

    printf("name associated with %s is %s\n", bdstr, hp->h_name);
```

DIAGNOSTICS

Error return status from **bt_gethostent**(), **bt_gethostbyname**(), and **bt_gethostbyaddr**() is indicated by return of a NULL pointer. The external integer *h_errno* may then be checked to see whether this is a temporary failure or an invalid or unknown host. The routine herror(3) can be used to print an error message describing the failure. If its argument *string* is non-NULL, it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

The variable *h_errno* can have the following values:

HOST_NOT_FOUND No such host is known.

NO_RECOVERY Some unexpected server failure was encountered. This is a non-recoverable error.

The **bt_getprotoent**(), **bt_getprotobyname**(), and **bt_getprotobynumber**() return NULL on EOF or error.

SEE ALSO

gethostbyaddr(3), gethostbyname(3), getprotobyname(3), getprotobynumber(3), herror(3), inet_aton(3), inet_ntoa(3)

HISTORY

libbluetooth first appeared in FreeBSD was ported to NetBSD 4.0 and extended by Iain Hibbert for Itronix, Inc.

AUTHORS

Maksim Yevmenkin ⟨m_evmenkin@yahoo.com⟩
Iain Hibbert

CAVEATS

The **bt_gethostent**() function reads the next line of /etc/bluetooth/hosts, opening the file if necessary.

The **bt_sethostent**() function opens and/or rewinds the /etc/bluetooth/hosts file.

The **bt_getprotoent**() function reads the next line of /etc/bluetooth/protocols, opening the file if necessary.

The **bt_setprotoent**() function opens and/or rewinds the /etc/bluetooth/protocols file.

BUGS

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it.

NAME

bm_comp, **bm_exec**, **bm_free** — Boyer-Moore string search

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <bm.h>

bm_pat *
bm_comp(u_char *pattern, size_t patlen, u_char freq[256]);

u_char *
bm_exec(bm_pat *pdesc, u_char *text, size_t len);

void
bm_free(bm_pat *pdesc);
```

DESCRIPTION

These routines implement an efficient mechanism to find an occurrence of a byte string within another byte string.

bm_comp() evaluates the *patlen* bytes starting at *pattern*, and returns a pointer to a structure describing them. The bytes referenced by *pattern* may be of any value.

The search takes advantage of the frequency distribution of the bytes in the text to be searched. If specified, *freq* should be an array of 256 values, with higher values indicating that the corresponding character occurs more frequently. (A less than optimal frequency distribution can only result in less than optimal performance, not incorrect results.) If *freq* is NULL, a system default table is used.

bm_exec() returns a pointer to the leftmost occurrence of the string given to **bm_comp**() within *text*, or NULL if none occurs. The number of bytes in *text* must be specified by *len*.

Space allocated for the returned description is discarded by calling **bm_free**() with the returned description as an argument.

The asymptotic speed of **bm_exec**() is O(len/patlen).

SEE ALSO

regexp(3), strstr(3)

Hume and Sunday, "Fast String Searching", *Software Practice and Experience*, Vol. 21, 11, pp. 1221-48, November 1991.

NAME

bsearch — binary search of a sorted table

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

void *
bsearch(const void *key, const void *base, size_t nmemb, size_t size,
    int (*compar) (const void *, const void *));
```

DESCRIPTION

The **bsearch**() function searches an array of *nmemb* objects, the initial member of which is pointed to by *base*, for a member that matches the object pointed to by *key*. The size of each member of the array is specified by *size*.

The contents of the array should be in ascending sorted order according to the comparison function referenced by *compar*. The *compar* routine is expected to have two arguments which point to the *key* object and to an array member, in that order, and should return an integer less than, equal to, or greater than zero if the *key* object is found, respectively, to be less than, to match, or be greater than the array member.

RETURN VALUES

The **bsearch**() function returns a pointer to a matching member of the array, or a null pointer if no match is found. If two members compare as equal, which member is matched is unspecified.

SEE ALSO

db(3), lsearch(3), qsort(3), tsearch(3)

STANDARDS

The **bsearch**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

memccpy, **memchr**, **memcmp**, **memcpy**, **memmem**, **memmove**, **memset** — byte string operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

void *
memchr(const void *b, int c, size_t len);

int
memcmp(const void *b1, const void *b2, size_t len);

void *
memccpy(void *dst, const void *src, int c, size_t len);

void *
memcpy(void *dst, const void *src, size_t len);

void *
memmem(const void *block, size_t blen, const void *pat, size_t plen);

void *
memmove(void *dst, const void *src, size_t len);

void *
memset(void *b, int c, size_t len);
```

DESCRIPTION

These functions operate on variable length strings of bytes. They do not check for terminating nul bytes as the routines listed in string(3) do.

See the specific manual pages for more information.

SEE ALSO

memccpy(3), memchr(3), memcmp(3), memcpy(3), memmem(3), memmove(3), memset(3)

STANDARDS

The functions **memchr**(), **memcmp**(), **memcpy**(), **memmove**(), and **memset**() conform to ANSI X3.159-1989 (“ANSI C89”).

HISTORY

The function **memccpy**() appeared in 4.3BSD.

NAME

bswap16, **bswap32**, **bswap64** — byte-order swapping functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <machine/bswap.h>

uint16_t
bswap16(uint16_t);

uint32_t
bswap32(uint32_t);

uint64_t
bswap64(uint64_t);
```

DESCRIPTION

The **bswap16**(), **bswap32**(), and **bswap64**() functions return the value of their argument with the bytes inverted. They can be used to convert 16, 32 or 64 bits integers from little to big endian, or vice-versa.

SEE ALSO

byteorder(3)

NAME

btowc — convert a single byte character to a wide character

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

wint_t
btowc(int c);
```

DESCRIPTION

The **btowc**() function converts a single byte character *c* in the initial shift state of the current locale to a corresponding wide character.

The behaviour of **btowc**() is affected by the LC_CTYPE category of the current locale.

RETURN VALUES

The **btowc**() function returns:

WEOF	If *c* is EOF or if (unsigned char) *c* does not correspond to a valid single byte character representation.
(otherwise)	A wide character corresponding to *c*.

ERRORS

No errors are defined.

SEE ALSO

mbrtowc(3), setlocale(3), wctob(3)

STANDARDS

The **btowc**() function conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”).

NAME

btree — btree database access method

SYNOPSIS

```
#include <sys/types.h>
#include <db.h>
```

DESCRIPTION

The routine **dbopen**() is the library interface to database files. One of the supported file formats is btree files. The general description of the database access methods is in dbopen(3), this manual page describes only the btree specific information.

The btree data structure is a sorted, balanced tree structure storing associated key/data pairs.

The btree access method specific data structure provided to **dbopen**() is defined in the ⟨db.h⟩ include file as follows:

```
typedef struct {
        u_long flags;
        u_int cachesize;
        int maxkeypage;
        int minkeypage;
        u_int psize;
        int (*compare)(const DBT *key1, const DBT *key2);
        size_t (*prefix)(const DBT *key1, const DBT *key2);
        int lorder;
} BTREEINFO;
```

The elements of this structure are as follows:

flags The flag value is specified by or'ing any of the following values:

R_DUP Permit duplicate keys in the tree, i.e. permit insertion if the key to be inserted already exists in the tree. The default behavior, as described in dbopen(3), is to overwrite a matching key when inserting a new key or to fail if the R_NOOVERWRITE flag is specified. The R_DUP flag is overridden by the R_NOOVERWRITE flag, and if the R_NOOVERWRITE flag is specified, attempts to insert duplicate keys into the tree will fail.

If the database contains duplicate keys, the order of retrieval of key/data pairs is undefined if the *get* routine is used, however, *seq* routine calls with the R_CURSOR flag set will always return the logical "first" of any group of duplicate keys.

cachesize A suggested maximum size (in bytes) of the memory cache. This value is *only* advisory, and the access method will allocate more memory rather than fail. Since every search examines the root page of the tree, caching the most recently used pages substantially improves access time. In addition, physical writes are delayed as long as possible, so a moderate cache can reduce the number of I/O operations significantly. Obviously, using a cache increases (but only increases) the likelihood of corruption or lost data if the system crashes while a tree is being modified. If *cachesize* is 0 (no size is specified) a default cache is used.

maxkeypage The maximum number of keys which will be stored on any single page. Not currently implemented.

minkeypage The minimum number of keys which will be stored on any single page. This value is used to determine which keys will be stored on overflow pages, i.e., if a key or data item is longer than the pagesize divided by the *minkeypage* value, it will be stored on overflow pages instead of in the page itself. If *minkeypage* is 0 (no minimum number of keys is specified) a value of 2 is used.

psize Page size is the size (in bytes) of the pages used for nodes in the tree. The minimum page size is 512 bytes and the maximum page size is 64K. If *psize* is 0 (no page size is specified) a page size is chosen based on the underlying file system I/O block size.

compare Compare is the key comparison function. It must return an integer less than, equal to, or greater than zero if the first key argument is considered to be respectively less than, equal to, or greater than the second key argument. The same comparison function must be used on a given tree every time it is opened. If *compare* is NULL (no comparison function is specified), the keys are compared lexically, with shorter keys considered less than longer keys.

prefix Prefix is the prefix comparison function. If specified, this routine must return the number of bytes of the second key argument which are necessary to determine that it is greater than the first key argument. If the keys are equal, the key length should be returned. Note, the usefulness of this routine is very data dependent, but, in some data sets can produce significantly reduced tree sizes and search times. If *prefix* is NULL (no prefix function is specified), *and* no comparison function is specified, a default lexical comparison routine is used. If *prefix* is NULL and a comparison routine is specified, no prefix comparison is done.

lorder The byte order for integers in the stored database metadata. The number should represent the order as an integer; for example, big endian order would be the number 4,321. If *lorder* is 0 (no order is specified) the current host order is used.

If the file already exists (and the O_TRUNC flag is not specified), the values specified for the parameters flags, lorder and psize are ignored in favor of the values used when the tree was created.

Forward sequential scans of a tree are from the least key to the greatest.

Space freed up by deleting key/data pairs from the tree is never reclaimed, although it is normally made available for reuse. This means that the btree storage structure is grow-only. The only solutions are to avoid excessive deletions, or to create a fresh tree periodically from a scan of an existing one.

Searches, insertions, and deletions in a btree will all complete in O lg base N where base is the average fill factor. Often, inserting ordered data into btrees results in a low fill factor. This implementation has been modified to make ordered insertion the best case, resulting in a much better than normal page fill factor.

ERRORS

The **btree** access method routines may fail and set *errno* for any of the errors specified for the library routine dbopen(3).

SEE ALSO

dbopen(3), hash(3), mpool(3), recno(3)

Douglas Comer, "The Ubiquitous B-tree", *ACM Comput. Surv.*, 11, 2, 121-138, June 1979.

Bayer and Unterauer, "Prefix B-trees", *ACM Transactions on Database Systems*, 1, Vol. 2, 11-26, March 1977.

D.E. Knuth, *The Art of Computer Programming Vol. 3: Sorting and Searching*, 471-480, 1968.

BUGS

Only big and little endian byte order is supported.

NAME

htonl, **htons**, **ntohl**, **ntohs** — convert values between host and network byte order

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <arpa/inet.h>

uint32_t
htonl(uint32_t host32);

uint16_t
htons(uint16_t host16);

uint32_t
ntohl(uint32_t net32);

uint16_t
ntohs(uint16_t net16);
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order.

On machines which have a byte order which is the same as the network order, these routines are defined as macros that expand to the value of their argument.

These routines are most often used in conjunction with Internet addresses and ports as returned by gethostbyname(3) and getservent(3).

SEE ALSO

gethostbyname(3), getservent(3)

STANDARDS

The **htonl**(), **htons**(), **ntohl**(), and **ntohs**() functions conform to IEEE Std 1003.1-2001 (“POSIX.1”). Their use of the fixed-width integer types *uint16_t* and *uint32_t* first appeared in X/Open Networking Services Issue 5 (“XNS5”).

HISTORY

The **byteorder** functions appeared in 4.2BSD.

BUGS

The ‘l’ and ‘s’ suffixes in the names are not meaningful in machines where long integers are not 32 bits.

NAME

bzero — write zeroes to a byte string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <strings.h>

void
bzero(void *b, size_t len);
```

DESCRIPTION

The **bzero**() function writes *len* zero bytes to the string *b*. If *len* is zero, **bzero**() does nothing.

SEE ALSO

memset(3), swab(3)

HISTORY

A **bzero**() function appeared in 4.3BSD.

NAME

cabs, cabsf − return a complex absolute value

SYNOPSIS

```
#include <complex.h>

double cabs(double complex z);
float cabsf(float complex z);
```

DESCRIPTION

These functions compute the complex absolute value (also called norm, modulus, or magnitude) of *z*.

RETURN VALUE

These functions return the complex absolute value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-2001, *<complex.h>*

COPYRIGHT

NAME

cacos, cacosf − complex arc cosine functions

SYNOPSIS

```
#include <complex.h>

double complex cacos(double complex z);
float complex cacosf(float complex z);
```

DESCRIPTION

These functions compute the complex arc cosine of *z*, with branch cuts outside the interval [-1, +1] along the real axis.

RETURN VALUE

These functions return the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0, pi] along the real axis.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ccos(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

cacosh, cacoshf − complex arc hyperbolic cosine functions

SYNOPSIS

#include <complex.h>

double complex cacosh(double complex *z***);**
float complex cacoshf(float complex *z***);**

DESCRIPTION

These functions compute the complex arc hyperbolic cosine of *z*, with a branch cut at values less than 1 along the real axis.

RETURN VALUE

These functions return the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis and in the interval [-*i*pi, +*i*pi] along the imaginary axis.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ccosh(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

carg, cargf − complex argument functions

SYNOPSIS

```
#include <complex.h>

double carg(double complex z);
float cargf(float complex z);
```

DESCRIPTION

These functions compute the argument (also called phase angle) of *z*, with a branch cut along the negative real axis.

RETURN VALUE

These functions return the value of the argument in the interval [-pi, +pi].

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cimag(), *conj*(), *cproj*(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

casin, casinf − complex arc sine functions

SYNOPSIS

```
#include <complex.h>

double complex casin(double complex z);
float complex casinf(float complex z);
```

DESCRIPTION

These functions compute the complex arc sine of *z*, with branch cuts outside the interval [-1, +1] along the real axis.

RETURN VALUE

These functions return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-pi/2, +pi/2] along the real axis.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

csin(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

casinh, casinhf − complex arc hyperbolic sine functions

SYNOPSIS

```
#include <complex.h>

double complex casinh(double complex z);
float complex casinhf(float complex z);
```

DESCRIPTION

These functions compute the complex arc hyperbolic sine of *z*, with branch cuts outside the interval [-*i*, +*i*] along the imaginary axis.

RETURN VALUE

These functions return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval [-*i*pi/2, +*i*pi/2] along the imaginary axis.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

csinh(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

catan, catanf − complex arc tangent functions

SYNOPSIS

#include <complex.h>

double complex catan(double complex *z***);**
float complex catanf(float complex *z***);**

DESCRIPTION

These functions compute the complex arc tangent of *z*, with branch cuts outside the interval [-*i*, +*i*] along the imaginary axis.

RETURN VALUE

These functions return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-pi/2, +pi/2] along the real axis.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctan(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

catanh, catanhf − complex arc hyperbolic tangent functions

SYNOPSIS

#include <complex.h>

double complex catanh(double complex *z***);**
float complex catanhf(float complex *z***);**

DESCRIPTION

These functions compute the complex arc hyperbolic tangent of *z*, with branch cuts outside the interval [-1, +1] along the real axis.

RETURN VALUE

These functions return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval [-*i*pi/2, +*i*pi/2] along the imaginary axis.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctanh(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

catclose — close message catalog

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <nl_types.h>

int
catclose(nl_catd catd);
```

DESCRIPTION

The **catclose**() function closes the message catalog specified by the argument *catd*.

SEE ALSO

gencat(1), catgets(3), catopen(3), nls(7)

STANDARDS

The **catclose**() function conforms to X/Open Portability Guide Issue 3 ("XPG3").

NAME

catgets — retrieve string from message catalog

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <nl_types.h>

char *
catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

DESCRIPTION

The **catgets**() function attempts to retrieve message *msg_id* of set *set_id* from the message catalog referenced by the descriptor *catd*. The argument *s* points to a default message which is returned if the function is unable to retrieve the specified message.

RETURN VALUES

If the specified message was retrieved successfully, **catgets**() returns a pointer to an internal buffer containing the message string; otherwise it returns *s*.

ERRORS

The **catgets**() function will fail if:

[EBADF]	The *catd* argument is not a valid message catalog descriptor open for reading.
[EINTR]	The operation was interrupted by a signal.
[ENOMSG]	The message identified by *set_id* and *msg_id* is not in the message catalog.

SEE ALSO

gencat(1), catclose(3), catopen(3), nls(7)

STANDARDS

The **catgets**() function conforms to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

Major Unix vendors are split over the adoption of the two most important message catalog specifications: catgets or gettext(3). The primary concern with the catgets interface is that every translatable string has to define a number (or a symbolic constant) which must correspond to the message in the catalog. Duplicate message IDs are not allowed. Constructing message catalogs is difficult.

NAME

catopen — open message catalog

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <nl_types.h>

nl_catd
catopen(const char *name, int oflag);
```

DESCRIPTION

The **catopen**() function opens the message catalog specified by *name* and returns a message catalog descriptor. If *name* contains a '/' then *name* specifies the full pathname for the message catalog, otherwise the value of the environment variable NLSPATH is used with *name* substituted for %N.

The *oflag* argument is reserved for future use and should be set to zero.

RETURN VALUES

Upon successful completion, **catopen**() returns a message catalog descriptor. Otherwise, (nl_catd) -1 is returned and *errno* is set to indicate the error.

ERRORS

[ENOMEM] Insufficient memory is available.

SEE ALSO

gencat(1), catclose(3), catgets(3), nls(7)

STANDARDS

The **catopen**() function conforms to X/Open Portability Guide Issue 3 ("XPG3").

NAME

ccos, ccosf − complex cosine functions

SYNOPSIS

```
#include <complex.h>

double complex ccos(double complex z);
float complex ccosf(float complex z);
```

DESCRIPTION

These functions compute the complex cosine of *z*.

RETURN VALUE

These functions return the complex cosine value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cacos(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

ccosh, ccoshf − complex hyperbolic cosine functions

SYNOPSIS

```
#include <complex.h>

double complex ccosh(double complex z);
float complex ccoshf(float complex z);
```

DESCRIPTION

These functions compute the complex hyperbolic cosine of *z*.

RETURN VALUE

These functions return the complex hyperbolic cosine value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cacosh(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

ceil, **ceilf** — round to smallest integral value greater than or equal to x

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
ceil(double x);

float
ceilf(float x);
```

DESCRIPTION

The **ceil**() and **ceilf**() functions return the smallest integral value greater than or equal to *x*.

SEE ALSO

abs(3), fabs(3), floor(3), ieee(3), math(3), rint(3)

STANDARDS

The **ceil**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

cexp, cexpf − complex exponential functions

SYNOPSIS

```
#include <complex.h>

double complex cexp(double complex z);
float complex cexpf(float complex z);
```

DESCRIPTION

These functions compute the complex exponent of *z*, defined as *e****z*.

RETURN VALUE

These functions return the complex exponential value of *z*.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clog(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

cgetent, **cgetset**, **cgetmatch**, **cgetcap**, **cgetnum**, **cgetstr**, **cgetustr**, **cgetfirst**, **cgetnext**, **cgetclose**, **cexpandtc** — capability database access routines

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
cgetent(char **buf, const char * const *db_array, const char *name);

int
cgetset(const char *ent);

int
cgetmatch(const char *buf, const char *name);

char *
cgetcap(char *buf, const char *cap, int type);

int
cgetnum(char *buf, const char *cap, long *num);

int
cgetstr(char *buf, const char *cap, char **str);

int
cgetustr(char *buf, const char *cap, char **str);

int
cgetfirst(char **buf, const char * const *db_array);

int
cgetnext(char **buf, const char * const *db_array);

int
cgetclose(void);

void
csetexpandtc(int expandtc);
```

DESCRIPTION

cgetent() extracts the capability *name* from the database specified by the NULL terminated file array *db_array* and returns a pointer to a malloc(3)'d copy of it in *buf*. **cgetent**() will first look for files ending in .db (see cap_mkdb(1)) before accessing the ASCII file.

buf must be retained through all subsequent calls to **cgetmatch**(), **cgetcap**(), **cgetnum**(), **cgetstr**(), and **cgetustr**(), but may then be free(3)'d.

On success 0 is returned, 1 if the returned record contains an unresolved "tc" expansion, −1 if the requested record couldn't be found, −2 if a system error was encountered (couldn't open/read a file, etc.) also setting *errno*, and −3 if a potential reference loop is detected (see "tc=name" comments below).

cgetset() enables the addition of a character buffer containing a single capability record entry to the capability database. Conceptually, the entry is added as the first "file" in the database, and is therefore searched first on the call to **cgetent**(). The entry is passed in *ent*. If *ent* is NULL, the current entry is removed from the database.

cgetset() must precede the database traversal. It must be called before the **cgetent**() call. If a sequential access is being performed (see below), it must be called before the first sequential access call (**cgetfirst**() or **cgetnext**()), or be directly preceded by a **cgetclose**() call. On success 0 is returned and −1 on failure.

cgetmatch() will return 0 if *name* is one of the names of the capability record *buf*, −1 if not.

cgetcap() searches the capability record *buf* for the capability *cap* with type *type*. A *type* is specified using any single character. If a colon (':') is used, an untyped capability will be searched for (see below for explanation of types). A pointer to the value of *cap* in *buf* is returned on success, NULL if the requested capability couldn't be found. The end of the capability value is signaled by a ':' or ASCII NUL (see below for capability database syntax).

cgetnum() retrieves the value of the numeric capability *cap* from the capability record pointed to by *buf*. The numeric value is returned in the *long* pointed to by *num*. 0 is returned on success, −1 if the requested numeric capability couldn't be found.

cgetstr() retrieves the value of the string capability *cap* from the capability record pointed to by *buf*. A pointer to a decoded, NUL terminated, malloc(3)'d copy of the string is returned in the *char* * pointed to by *str*. The number of characters in the decoded string not including the trailing NUL is returned on success, −1 if the requested string capability couldn't be found, −2 if a system error was encountered (storage allocation failure).

cgetustr() is identical to **cgetstr**() except that it does not expand special characters, but rather returns each character of the capability string literally.

cgetfirst(), **cgetnext**(), comprise a function group that provides for sequential access of the NULL pointer terminated array of file names, *db_array*. **cgetfirst**() returns the first record in the database and resets the access to the first record. **cgetnext**() returns the next record in the database with respect to the record returned by the previous **cgetfirst**() or **cgetnext**() call. If there is no such previous call, the first record in the database is returned. Each record is returned in a malloc(3)'d copy pointed to by *buf*. "tc" expansion is done (see "tc=name" comments below).

Upon completion of the database 0 is returned, 1 is returned upon successful return of record with possibly more remaining (we haven't reached the end of the database yet), 2 is returned if the record contains an unresolved "tc" expansion, −1 is returned if an system error occurred, and −2 is returned if a potential reference loop is detected (see "tc=name" comments below). Upon completion of database (0 return) the database is closed.

cgetclose() closes the sequential access and frees any memory and file descriptors being used. Note that it does not erase the buffer pushed by a call to **cgetset**().

CAPABILITY DATABASE SYNTAX

Capability databases are normally ASCII and may be edited with standard text editors. Blank lines and lines beginning with a '#' are comments and are ignored. Lines ending with a '\' indicate that the next line is a continuation of the current line; the '\' and following newline are ignored. Long lines are usually continued onto several physical lines by ending each line except the last with a '\'.

Capability databases consist of a series of records, one per logical line. Each record contains a variable number of ':'-separated fields (capabilities). Empty fields consisting entirely of white space characters (spaces and tabs) are ignored.

The first capability of each record specifies its names, separated by '|' characters. These names are used to reference records in the database. By convention, the last name is usually a comment and is not intended as a lookup tag. For example, the *vt100* record from the termcap database begins:

```
d0|vt100|vt100-am|vt100am|dec vt100:
```

giving four names that can be used to access the record.

The remaining non-empty capabilities describe a set of (name, value) bindings, consisting of a name optionally followed by a typed value:

name	typeless [boolean] capability *name* is present [true]
name*T*value	capability (*name*, *T*) has value *value*
name@	no capability *name* exists
name*T*@	capability (*name*, *T*) does not exist

Names consist of one or more characters. Names may contain any character except ‘:’, but it’s usually best to restrict them to the printable characters and avoid use of graphics like ‘#’, ‘=’, ‘%’, ‘@’, etc.

Types are single characters used to separate capability names from their associated typed values. Types may be any character except a ‘:’. Typically, graphics like ‘#’, ‘=’, ‘%’, etc. are used. Values may be any number of characters and may contain any character except ‘:’.

CAPABILITY DATABASE SEMANTICS

Capability records describe a set of (name, value) bindings. Names may have multiple values bound to them. Different values for a name are distinguished by their *types*. **cgetcap**() will return a pointer to a value of a name given the capability name and the type of the value.

The types ‘#’ and ‘=’ are conventionally used to denote numeric and string typed values, but no restriction on those types is enforced. The functions **cgetnum**() and **cgetstr**() can be used to implement the traditional syntax and semantics of ‘#’ and ‘=’. Typeless capabilities are typically used to denote boolean objects with presence or absence indicating truth and false values respectively. This interpretation is conveniently represented by:

```
(getcap(buf, name, ':') != NULL)
```

A special capability, "tc=name", is used to indicate that the record specified by *name* should be substituted for the "tc" capability. "tc" capabilities may interpolate records which also contain "tc" capabilities and more than one "tc" capability may be used in a record. A "tc" expansion scope (i.e. where the argument is searched for) contains the file in which the "tc" is declared and all subsequent files in the file array.

csetexpandtc() can be used to control if
" tc expansion is performed or not.

When a database is searched for a capability record, the first matching record in the search is returned. When a record is scanned for a capability, the first matching capability is returned; the capability ":nameT@:" will hide any following definition of a value of type *T* for *name*; and the capability ":name@:" will prevent any following values of *name* from being seen.

These features combined with "tc" capabilities can be used to generate variations of other databases and records by either adding new capabilities, overriding definitions with new definitions, or hiding following definitions via ‘@’ capabilities.

EXAMPLES

```
example|an example of binding multiple values to names:\
        :foo%bar:foo^blah:foo@:\
        :abc%xyz:abc^frap:abc$@:\
        :tc=more:
```

The capability *foo* has two values bound to it (*bar* of type ‘%’ and *blah* of type ‘^’) and any other value bindings are hidden. The capability *abc* also has two values bound but only a value of type ‘$’ is prevented from being defined in the capability record more.

```
file1:
        new|new_record|a modification of "old":\
                :fript=bar:who-cares@:tc=old:blah:tc=extensions:
file2:
        old|old_record|an old database record:\
                :fript=foo:who-cares:glork#200:
```

The records are extracted by calling **cgetent**() with *file1* preceding *file2*. In the capability record *new* in *file1*, "fript=bar" overrides the definition of "fript=foo" interpolated from the capability record *old* in *file2*, "who-cares@" prevents the definition of any who-cares definitions in *old* from being seen, "glork#200" is inherited from *old*, and *blah* and anything defined by the record extensions is added to those definitions in *old*. Note that the position of the "fript=bar" and "who-cares@" definitions before "tc=old" is important here. If they were after, the definitions in *old* would take precedence.

CGETNUM AND CGETSTR SYNTAX AND SEMANTICS

Two types are predefined by **cgetnum**() and **cgetstr**():

name#number	numeric capability *name* has value *number*
name=string	string capability *name* has value *string*
name#@	the numeric capability *name* does not exist
name=@	the string capability *name* does not exist

Numeric capability values may be given in one of three numeric bases. If the number starts with either '0x' or '0X' it is interpreted as a hexadecimal number (both upper and lower case a-f may be used to denote the extended hexadecimal digits). Otherwise, if the number starts with a '0' it is interpreted as an octal number. Otherwise the number is interpreted as a decimal number.

String capability values may contain any character. Non-printable ASCII codes, new lines, and colons may be conveniently represented by the use of escape sequences:

^X	('*X*' & 037)	control-*X*
\b, \B	(ASCII 010)	backspace
\t, \T	(ASCII 011)	tab
\n, \N	(ASCII 012)	line feed (newline)
\f, \F	(ASCII 014)	form feed
\r, \R	(ASCII 015)	carriage return
\e, \E	(ASCII 027)	escape
\c, \C	(:)	colon
\\	(\)	back slash
\^	(^)	caret
nnn	(ASCII octal *nnn*)	

A '\' followed by up to three octal digits directly specifies the numeric code for a character. The use of ASCII NULs, while easily encoded, causes all sorts of problems and must be used with care since NULs are typically used to denote the end of strings; many applications use '\200' to represent a NUL.

DIAGNOSTICS

cgetent(), **cgetset**(), **cgetmatch**(), **cgetnum**(), **cgetstr**(), **cgetustr**(), **cgetfirst**(), and **cgetnext**() return a value greater than or equal to 0 on success and a value less than 0 on failure. **cgetcap**() returns a character pointer on success and a NULL on failure.

cgetclose(), **cgetent**(), **cgetfirst**(), and **cgetnext**() may fail and set *errno* for any of the errors specified for the library functions: fopen(3), fclose(3), open(2), and close(2).

cgetent(), **cgetset**(), **cgetstr**(), and **cgetustr**() may fail and set *errno* as follows:

[ENOMEM] No memory to allocate.

SEE ALSO

cap_mkdb(1), malloc(3)

BUGS

Colons (':') can't be used in names, types, or values.

There are no checks for "tc=name" loops in **cgetent**().

The buffer added to the database by a call to **cgetset**() is not unique to the database but is rather prepended to any database used.

NAME

cimag, cimagf − complex imaginary functions

SYNOPSIS

```
#include <complex.h>

double cimag(double complex z);
float cimagf(float complex z);
```

DESCRIPTION

These functions compute the imaginary part of *z*.

RETURN VALUE

These functions return the imaginary part value (as a real).

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

For a variable *z* of complex type:

```
z == creal(z) + cimag(z)*I
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg(), *conj*(), *cproj*(), *creal*(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

clock — determine processor time used

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <time.h>

clock_t
clock(void);
```

DESCRIPTION

The **clock**() function determines the amount of processor time used since the invocation of the calling process, measured in CLOCKS_PER_SECs.

RETURN VALUES

The **clock**() function returns the amount of time used unless an error occurs, in which case the return value is −1.

SEE ALSO

getrusage(2)

STANDARDS

The **clock**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

clog, clogf − complex natural logarithm functions

SYNOPSIS

```
#include <complex.h>

double complex clog(double complex z);
float complex clogf(float complex z);
```

DESCRIPTION

These functions compute the complex natural (base *e*) logarithm of *z*, with a branch cut along the negative real axis.

RETURN VALUE

These functions return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval [-*i*pi, +*i*pi] along the imaginary axis.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cexp(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

closefrom — delete many descriptors

SYNOPSIS

```
#include <unistd.h>

int
closefrom(int fd);
```

DESCRIPTION

The **closefrom**() call deletes all descriptors numbered *fd* and higher from the per-process file descriptor table. It is effectively the same as calling close(2) on each descriptor.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

closefrom() will fail if:

[EBADF] *fd* is invalid.

[EINTR] An interrupt was received.

SEE ALSO

close(2)

NAME

com_err, **com_err_va**, **error_message**, **error_table_name**, **init_error_table**, **set_com_err_hook**, **reset_com_err_hook** — common error display library

LIBRARY

Common Error Library (libcom_err, -lcom_err)

SYNOPSIS

```
#include <stdio.h>
#include <stdarg.h>
#include <krb5/com_err.h>
#include "XXX_err.h"
```

typedef void (*errf)(const char *, long, const char *, ...);

void
com_err(*const char *whoami*, *long code*, *const char *format*, *...*);

void
com_err_va(*const char *whoami*, *long code*, *const char *format*, *...*);

*const char **
error_message(*long code*);

*const char **
error_table_name(*int num*);

int
init_error_table(*const char **msgs*, *long base*, *int count*);

errf
set_com_err_hook(*errf func*);

errf
reset_com_err_hook();

void
add_to_error_table(*struct et_list *new_table*);

DESCRIPTION

The **com_err** library provides a common error-reporting mechanism for defining and accessing error codes and descriptions for application software packages. Error descriptions are defined in a table and error codes are used to index the table. The error table, the descriptions and the error codes are generated using compile_et(1).

The error table is registered with the **com_err** library by calling its initialisation function defined in its header file. The initialisation function is generally defined as **initialize_<name>_error_table**(), where *name* is the name of the error table.

Any variable which is to contain an error code should be declared *<name>_error_number* where *name* is the name of the error table.

FUNCTIONS

The following functions are available to the application developer:

com_err(*whoami*, *code*, *format*, *...*)

Displays an error message on standard error composed of the *whoami* string, which should specify the program name, followed by an error message generated from *code*, and a string produced

using the printf(3) *format* string and any following arguments. If *format* is NULL, the formatted message will not be printed. The argument *format* may not be omitted.

com_err_va(*whoami*, *code*, *format*, *va_list args*)
This routine provides an interface, equivalent to **com_err**(), which may be used by higher-level variadic functions (functions which accept variable numbers of arguments).

error_message(*code*)
Returns the character string error message associate with *code*. If *code is associated with an unknown error table, or if code* is associated with a known error table but is not in the table, a string of the form 'Unknown code XXXX NN' is returned, where XXXX is the error table name produced by reversing the compaction performed on the error table number implied by that error code, and NN is the offset from that base value.

Although this routine is available for use when needed, its use should be left to circumstances which render **com_err**() unusable.

error_table_name(*num*)
Convert a machine-independent error table number *num* into an error table name.

init_error_table(*msgs*, *base*, *count*)
Initialise the internal error table with the array of character string error messages in *msgs* of length *count*. The error codes are assigned incrementally from *base*. This function is useful for using the error-reporting mechanism with custom error tables that have not been generated with compile_et(1). Although this routine is available for use when needed, its use should be restricted.

set_com_err_hook(*func*)
Provides a hook into the **com_err** library to allow the routine *func* to be dynamically substituted for **com_err**(). After **set_com_err_hook**()
has been called, calls to **com_err**() will turn into calls to the new hook routine. This function is intended to be used in daemons to use a routine which calls syslog(3), or in a window system application to pop up a dialogue box.

reset_com_err_hook()
Turns off the hook set in **set_com_err_hook**().

add_to_error_table(*new_table*)
Add the error table, its messages strings and error codes in *new_table* to the internal error table.

EXAMPLES

The following is an example using the table defined in compile_et(1):

```
#include <stdio.h>
#include <stdarg.h>
#include <syslog.h>

#include "test_err.h"

void
hook(const char *whoami, long code,
        const char *format, va_list args)
{
        char buffer[BUFSIZ];
        static int initialized = 0;
```

```
                if (!initialized) {
                        openlog(whoami, LOG_NOWAIT, LOG_DAEMON);
                        initialized = 1;
                }
                vsprintf(buffer, format, args);
                syslog(LOG_ERR, "%s %s", error_message(code), buffer);
        }

        int
        main(int argc, char *argv[])
        {
                char *whoami = argv[0];

                initialize_test_error_table();
                com_err(whoami, TEST_INVAL, "before hook");
                set_com_err_hook(hook);
                com_err(whoami, TEST_IO, "after hook");
                return (0);
        }
```

SEE ALSO

compile_et(1)

NAME

confstr — get string-valued configurable variables

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

size_t
confstr(int name, char *buf, size_t len);
```

DESCRIPTION

This interface is obsoleted by sysctl**(3).**

The **confstr**() function provides a method for applications to get configuration defined string values.

The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file <unistd.h>. The *len* argument specifies the size of the buffer referenced by the argument *buf*. If *len* is non-zero, *buf* is a non-null pointer, and *name* has a value, up to *len* − 1 bytes of the value are copied into the buffer *buf*. The copied value is always null terminated.

The available values are as follows:

_CS_PATH
: Return a value for the PATH environment variable that finds all the standard utilities.

RETURN VALUES

If the call to **confstr** is not successful, 0 is returned and *errno* is set appropriately. Otherwise, if the variable does not have a configuration defined value, 0 is returned and *errno* is not modified. Otherwise, the buffer size needed to hold the entire configuration-defined value is returned. If this size is greater than the argument *len*, the string in *buf* was truncated.

ERRORS

The **confstr** function may fail and set *error* for any of the errors specified for the library functions malloc(3) and sysctl(3).

In addition, the following errors may be reported:

[EINVAL] The value of the *name* argument is invalid.

SEE ALSO

sysctl(3)

STANDARDS

The **confstr** function conforms to IEEE Std 1003.2-1992 (“POSIX.2”).

HISTORY

The **confstr** function first appeared in 4.4BSD.

BUGS

The standards require us to return 0 both on errors, and when the value is not set.

NAME

conj, conjf − complex conjugate functions

SYNOPSIS

```
#include <complex.h>

double complex conj(double complex z);
float complex conjf(float complex z);
```

DESCRIPTION

These functions compute the complex conjugate of *z*, by reversing the sign of its imaginary part.

RETURN VALUE

These functions return the complex conjugate value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg(), *cimag*(), *cproj*(), *creal*(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

cos, **cosf** — cosine function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
cos(double x);

float
cosf(float x);
```

DESCRIPTION

The **cos**() and **cosf**() functions compute the cosine of *x* (measured in radians). A large magnitude argument may yield a result with little or no significance. For a discussion of error due to roundoff, see math(3).

RETURN VALUES

The **cos**() function returns the cosine value.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cosh(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **cos**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

cosh, **coshf** — hyperbolic cosine function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
cosh(double x);

float
coshf(float x);
```

DESCRIPTION

The **cosh**() and **coshf**() functions compute the hyperbolic cosine of *x*.

RETURN VALUES

If the magnitude of x is too large, **cosh**(*x*) and **coshf**(*x*) return Inf and sets the global variable *errno* to ERANGE.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cos(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **cosh**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

cpow, cpowf − complex power functions

SYNOPSIS

```
#include <complex.h>

double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
```

DESCRIPTION

These functions compute the complex power function *x****y*, with a branch cut for the first parameter along the negative real axis.

RETURN VALUE

These functions return the complex power function value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cabs(), *csqrt*(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

creal, crealf − complex real functions

SYNOPSIS

```
#include <complex.h>

double creal(double complex z);
float crealf(float complex z);
```

DESCRIPTION

These functions compute the real part of *z*.

RETURN VALUE

These functions return the real part value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

For a variable *z* of type **complex**:

```
z == creal(z) + cimag(z)*I
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg(), *cimag*(), *conj*(), *cproj*(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

creat — create a new file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <fcntl.h>

int
creat(const char *path, mode_t mode);
```

DESCRIPTION

This interface is made obsolete by: open(2).

creat() is the same as:

```
open(path, O_CREAT | O_TRUNC | O_WRONLY, mode);
```

SEE ALSO

open(2)

STANDARDS

The **creat**() function call conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

The **creat**() function call appeared in Version 6 AT&T UNIX.

NAME

crypt, **setkey**, **encrypt**, **des_setkey**, **des_cipher** — password encryption

LIBRARY

Crypt Library (libcrypt, −lcrypt)

SYNOPSIS

```
#include <unistd.h>

char
*crypt(const char *key, const char *setting);

int
encrypt(char *block, int flag);

int
des_setkey(const char *key);

int
des_cipher(const char *in, char *out, long salt, int count);

#include <stdlib.h>

int
setkey(const char *key);
```

DESCRIPTION

The **crypt**() function performs password encryption. The encryption scheme used by **crypt**() is dependent upon the contents of the NUL-terminated string *setting*. If it begins with a string character (‘$’) and a number then a different algorithm is used depending on the number. At the moment a ‘$1’ chooses MD5 hashing and a ‘$2’ chooses Blowfish hashing; see below for more information. If *setting* begins with the ‘‘_’’ character, DES encryption with a user specified number of perturbations is selected. If *setting* begins with any other character, DES encryption with a fixed number of perturbations is selected.

DES encryption

The DES encryption scheme is derived from the NBS Data Encryption Standard. Additional code has been added to deter key search attempts and to use stronger hashing algorithms. In the DES case, the second argument to **crypt**() is a character array, 9 bytes in length, consisting of an underscore (‘‘_’’) followed by 4 bytes of iteration count and 4 bytes of salt. Both the iteration *count* and the *salt* are encoded with 6 bits per character, least significant bits first. The values 0 to 63 are encoded by the characters ‘‘./0-9A-Za-z’’, respectively.

The *salt* is used to induce disorder in to the DES algorithm in one of 16777216 possible ways (specifically, if bit *i* of the *salt* is set then bits *i* and *i+24* are swapped in the DES ‘‘E’’ box output). The *key* is divided into groups of 8 characters (a short final group is null-padded) and the low-order 7 bits of each character (56 bits per group) are used to form the DES key as follows: the first group of 56 bits becomes the initial DES key. For each additional group, the XOR of the group bits and the encryption of the DES key with itself becomes the next DES key. Then the final DES key is used to perform *count* cumulative encryptions of a 64-bit constant. The value returned is a NUL-terminated string, 20 bytes in length, consisting of the *setting* followed by the encoded 64-bit encryption.

For compatibility with historical versions of crypt(3), the *setting* may consist of 2 bytes of salt, encoded as above, in which case an iteration *count* of 25 is used, fewer perturbations of DES are available, at most 8 characters of *key* are used, and the returned value is a NUL-terminated string 13 bytes in length.

The functions **encrypt**(), **setkey**(), **des_setkey**() and **des_cipher**() allow limited access to the DES algorithm itself. The *key* argument to **setkey**() is a 64 character array of binary values (numeric 0 or 1). A 56-bit key is derived from this array by dividing the array into groups of 8 and ignoring the last bit in each group.

The **encrypt**() argument *block* is also a 64 character array of binary values. If the value of *flag* is 0, the argument *block* is encrypted, otherwise it is decrypted. The encryption or decryption is returned in the original array *block* after using the key specified by **setkey**() to process it.

The **des_setkey**() and **des_cipher**() functions are faster but less portable than **setkey**() and **encrypt**(). The argument to **des_setkey**() is a character array of length 8. The *least* significant bit in each character is ignored and the next 7 bits of each character are concatenated to yield a 56-bit key. The function **des_cipher**() encrypts (or decrypts if *count* is negative) the 64-bits stored in the 8 characters at *in* using abs(3) of *count* iterations of DES and stores the 64-bit result in the 8 characters at *out*. The *salt* specifies perturbations to DES as described above.

MD5 encryption

For the MD5 encryption scheme, the version number (in this case ‘‘1’’), *salt* and the hashed password are separated by the ‘‘$’’ character. A valid password looks like this:

‘‘$1$2qGr5PPQ$eT08WBFev3RPLNChixg0H.’’.

The entire password string is passed as *setting* for interpretation.

Blowfish crypt

The Blowfish version of crypt has 128 bits of *salt* in order to make building dictionaries of common passwords space consuming. The initial state of the Blowfish cipher is expanded using the *salt* and the *password* repeating the process a variable number of rounds, which is encoded in the password string. The maximum password length is 72. The final Blowfish password entry is created by encrypting the string

“OrpheanBeholderScryDoubt”

with the Blowfish state 64 times.

The version number, the logarithm of the number of rounds and the concatenation of salt and hashed password are separated by the ‘$’ character. An encoded ‘8’ would specify 256 rounds. A valid Blowfish password looks like this:

“$2a$12$eIAq8PR8sIUnJ1HaohxX2O9x9Qlm2vK97LJ5dsXdmB.eXF42qjchC”.

The whole Blowfish password string is passed as *setting* for interpretation.

RETURN VALUES

The function **crypt**() returns a pointer to the encrypted value on success and NULL on failure. The functions **setkey**(), **encrypt**(), **des_setkey**(), and **des_cipher**() return 0 on success and 1 on failure. Historically, the functions **setkey**() and **encrypt**() did not return any value. They have been provided return values primarily to distinguish implementations where hardware support is provided but not available or where the DES encryption is not available due to the usual political silliness.

SEE ALSO

login(1), passwd(1), pwhash(1), getpass(3), md5(3), passwd(5), passwd.conf(5)

Wayne Patterson, *Mathematical Cryptology for Computer Scientists and Mathematicians*, ISBN 0-8476-7438-X, 1987.

R. Morris and Ken Thompson, "Password Security: A Case History", *Communications of the ACM*, vol. 22, pp. 594-597, Nov. 1979.

M.E. Hellman, "DES will be Totally Insecure within Ten Years", *IEEE Spectrum*, vol. 16, pp. 32-39, July 1979.

HISTORY

A rotor-based **crypt**() function appeared in Version 6 AT&T UNIX. The current style **crypt**() first appeared in Version 7 AT&T UNIX.

BUGS

Dropping the *least* significant bit in each character of the argument to **des_setkey**() is ridiculous.

The **crypt**() function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to **crypt**() will modify the same object.

NAME

crypto − OpenSSL cryptographic library

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

DESCRIPTION

The OpenSSL **crypto** library implements a wide range of cryptographic algorithms used in various Internet standards. The services provided by this library are used by the OpenSSL implementations of SSL, TLS and S/MIME, and they have also been used to implement SSH, OpenPGP, and other cryptographic standards.

OVERVIEW

libcrypto consists of a number of sub-libraries that implement the individual algorithms.

The functionality includes symmetric encryption, public key cryptography and key agreement, certificate handling, cryptographic hash functions and a cryptographic pseudo-random number generator.

SYMMETRIC CIPHERS

openssl_blowfish (3), *cast* (3), *openssl_des* (3), *idea* (3), *rc2* (3), *openssl_rc4* (3), *rc5* (3)

PUBLIC KEY CRYPTOGRAPHY AND KEY AGREEMENT

openssl_dsa (3), *openssl_dh* (3), *openssl_rsa* (3)

CERTIFICATES

openssl_x509 (3), *x509v3* (3)

AUTHENTICATION CODES, HASH FUNCTIONS

openssl_hmac (3), *md2* (3), *md4* (3), *openssl_md5* (3), *openssl_mdc2* (3), *openssl_ripemd* (3), *openssl_sha* (3)

AUXILIARY FUNCTIONS

openssl_err (3), *openssl_threads* (3), *openssl_rand* (3), *OPENSSL_VERSION_NUMBER* (3)

INPUT/OUTPUT, DATA ENCODING

asn1 (3), *openssl_bio* (3), *openssl_evp* (3), *openssl_pem* (3), *openssl_pkcs7* (3), *openssl_pkcs12* (3)

INTERNAL FUNCTIONS

openssl_bn (3), *openssl_buffer* (3), *openssl_lhash* (3), *objects* (3), *stack* (3), *txt_db* (3)

NOTES

Some of the newer functions follow a naming convention using the numbers **0** and **1**. For example the functions:

```
int X509_CRL_add0_revoked(X509_CRL *crl, X509_REVOKED *rev);
int X509_add1_trust_object(X509 *x, ASN1_OBJECT *obj);
```

The **0** version uses the supplied structure pointer directly in the parent and it will be freed up when the parent is freed. In the above example **crl** would be freed but **rev** would not.

The **1** function uses a copy of the supplied structure pointer (or in some cases increases its link count) in the parent and so both (**x** and **obj** above) should be freed up.

SEE ALSO

openssl (1), *ssl* (3)

NAME

csin, csinf − complex sine functions

SYNOPSIS

```
#include <complex.h>

double complex csin(double complex z);
float complex csinf(float complex z);
```

DESCRIPTION

These functions compute the complex sine of *z*.

RETURN VALUE

These functions return the complex sine value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

casin(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

csinh, csinhf − complex hyperbolic sine functions

SYNOPSIS

```
#include <complex.h>

double complex csinh(double complex z);
float complex csinhf(float complex z);
```

DESCRIPTION

These functions compute the complex hyperbolic sine of *z*.

RETURN VALUE

These functions return the complex hyperbolic sine value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

casinh(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

csqrt, csqrtf − complex square root functions

SYNOPSIS

```
#include <complex.h>

double complex csqrt(double complex z);
float complex csqrtf(float complex z);
```

DESCRIPTION

These functions compute the complex square root of *z*, with a branch cut along the negative real axis.

RETURN VALUE

These functions return the complex square root value, in the range of the right half-plane (including the imaginary axis).

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cabs(), *cpow*(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

ctan, ctanf − complex tangent functions

SYNOPSIS

```
#include <complex.h>

double complex ctan(double complex z);
float complex ctanf(float complex z);
```

DESCRIPTION

These functions compute the complex tangent of *z*.

RETURN VALUE

These functions return the complex tangent value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catan(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

ctanh, ctanhf − complex hyperbolic tangent functions

SYNOPSIS

```
#include <complex.h>

double complex ctanh(double complex z);
float complex ctanhf(float complex z);
```

DESCRIPTION

These functions compute the complex hyperbolic tangent of *z*.

RETURN VALUE

These functions return the complex hyperbolic tangent value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catanh(), the Base Definitions volume of IEEE Std 1003.1-2001, <*complex.h*>

COPYRIGHT

NAME

ctermid — generate terminal pathname

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

char *
ctermid(char *buf);
```

DESCRIPTION

The **ctermid**() function generates a string, that, when used as a pathname, refers to the current controlling terminal of the calling process.

If *buf* is the NULL pointer, a pointer to a static area is returned. Otherwise, the pathname is copied into the memory referenced by *buf*. The argument *buf* is assumed to point to an array at least L_ctermid bytes long (as defined in the include file ⟨stdio.h⟩).

The current implementation simply returns /dev/tty.

RETURN VALUES

Upon successful completion, a non-NULL pointer is returned. Otherwise, a NULL pointer is returned and the global variable *errno* is set to indicate the error.

ERRORS

The current implementation detects no error conditions.

SEE ALSO

ttyname(3)

STANDARDS

The **ctermid**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

BUGS

By default the **ctermid**() function writes all information to an internal static object. Subsequent calls to **ctermid**() will modify the same object.

NAME

asctime, **asctime_r**, **ctime**, **ctime_r**, **difftime**, **gmtime**, **gmtime_r**, **localtime**, **localtime_r**, **mktime** — convert date and time to ASCII

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <time.h>
extern char *tzname[2];

char *
ctime(const time_t *clock);

char *
ctime_r(const time_t *clock, char *buf);

double
difftime(time_t time1, time_t time0);

char *
asctime(const struct tm *tm);

char *
asctime_r(const struct tm restrict tm, char * restrict buf);

struct tm *
localtime(const time_t *clock);

struct tm *
localtime_r(const time_t * restrict clock, struct tm * restrict result);

struct tm *
gmtime(const time_t *clock);

struct tm *
gmtime_r(const time_t * restrict clock, struct tm * restrict result);

time_t
mktime(struct tm *tm);
```

DESCRIPTION

ctime() converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 UTC, 1970-01-01, and returns a pointer to a 26-character string of the form

```
Thu Nov 24 18:22:48 1986\n\0
```

All the fields have constant width.

The **ctime_r**() function provides the same functionality as **ctime**() differing in that the caller must supply a buffer area *buf* with a size of at least 26 bytes, in which the result is stored.

localtime() and **gmtime**() return pointers to *tm* structures, described below. **localtime**() corrects for the time zone and any time zone adjustments (such as Daylight Saving Time in the U.S.A.). After filling in the *tm* structure, **localtime**() sets the *tm_isdst*'th element of *tzname* to a pointer to an ASCII string that's the time zone abbreviation to be used with **localtime**()'s return value.

gmtime() converts to Coordinated Universal Time.

The **gmtime_r**() and **localtime_r**() functions provide the same functionality as **gmtime**() and **localtime**() differing in that the caller must supply a buffer area *result* in which the result is stored; also, **localtime_r**() does not imply initialization of the local time conversion information; the application may need to do so by calling tzset(3).

asctime() converts a time value contained in a *tm* structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

The **asctime_r**() function provides the same functionality as **asctime**() differing in that the caller must supply a buffer area *buf* with a size of at least 26 bytes, in which the result is stored.

mktime() converts the broken-down time, expressed as local time, in the structure pointed to by *tm* into a calendar time value with the same encoding as that of the values returned by the time(3) function. The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to their normal ranges. (A positive or zero value for *tm_isdst* causes **mktime**() to presume initially that summer time (for example, Daylight Saving Time in the U.S.A.) respectively, is or is not in effect for the specified time. A negative value for *tm_isdst* causes the **mktime**() function to attempt to divine whether summer time is in effect for the specified time.) On successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to their normal ranges; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined. **mktime**() returns the specified calendar time; if the calendar time cannot be represented, it returns -1.

difftime() returns the difference between two calendar times, (*time1* - *time0*), expressed in seconds.

The structure (of type) *struct tm* includes the following fields:

```
int tm_sec;    /* seconds after the minute [0,61] */
int tm_min;    /* minutes after the hour [0,59] */
int tm_hour;   /* hours since midnight [0,23] */
int tm_mday;   /* day of the month [1,31] */
int tm_mon;    /* months since January [0,11] */
int tm_year;   /* years since 1900 */
int tm_wday;   /* day of week [0,6] (Sunday = 0) */
int tm_yday;   /* day of year [0,365] (Jan 1 = 0) */
int tm_isdst;  /* daylight savings flag */
long tm_gmtoff;        /* offset from UTC in seconds */
char *tm_zone; /* abbreviation of timezone name */
```

The *tm_zone* and *tm_gmtoff* fields exist, and are filled in, only if arrangements to do so were made when the library containing these functions was created. There is no guarantee that these fields will continue to exist in this form in future releases of this code.

tm_isdst is non-zero if summer time is in effect.

tm_gmtoff is the offset (in seconds) of the time represented from UTC, with positive values indicating east of the Prime Meridian.

FILES

/etc/localtime	local time zone file
/usr/share/zoneinfo	time zone information directory
/usr/share/zoneinfo/posixrules	used with POSIX-style TZ's
/usr/share/zoneinfo/GMT	for UTC leap seconds

If /usr/share/zoneinfo/GMT is absent, UTC leap seconds are loaded from /usr/share/zoneinfo/posixrules.

SEE ALSO

`getenv`(3), `strftime`(3), `time`(3), `tzset`(3), `tzfile`(5)

STANDARDS

The **`ctime`**(), **`difftime`**(), **`asctime`**(), **`localtime`**(), **`gmtime`**() and **`mktime`**() functions conform to ANSI X3.159-1989 ("ANSI C89") The **`ctime_r`**(), **`asctime_r`**(), **`localtime_r`**() and **`gmtime_r`**() functions conform to IEEE Std 1003.1c-1995 ("POSIX.1").

NOTES

The return values point to static data; the data is overwritten by each call. The *`tm_zone`* field of a returned *struct tm* points to a static array of characters, which will also be overwritten at the next call (and by calls to `tzset`(3)).

Avoid using out-of-range values with **`mktime`**() when setting up lunch with promptness sticklers in Riyadh.

NAME

isalpha, **isupper**, **islower**, **isdigit**, **isxdigit**, **isalnum**, **isspace**, **ispunct**, **isprint**, **isgraph**, **iscntrl**, **isblank**, **toupper**, **tolower**, — character classification and mapping functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>
int c
```

isalpha(*c*);

isupper(*c*);

islower(*c*);

isdigit(*c*);

isxdigit(*c*);

isalnum(*c*);

isspace(*c*);

ispunct(*c*);

isprint(*c*);

isgraph(*c*);

iscntrl(*c*);

isblank(*c*);

toupper(*c*);

tolower(*c*);

DESCRIPTION

The above functions perform character tests and conversions on the integer *c*.

See the specific manual pages for information about the test or conversion performed by each function.

EXAMPLES

To print an upper-case version of a string on stdout, use the following code:

```
const char *s = ...;
while (*s != '\0') {
    putchar(toupper((int)(unsigned char)*s));
    s++;
}
```

SEE ALSO

isalnum(3), isalpha(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), tolower(3), toupper(3), ascii(7)

STANDARDS

These functions, with the exception of **isblank**(), conform to ANSI X3.159-1989 ("ANSI C89").

CAVEATS

The first argument of these functions is of type *int*, but only a very restricted subset of values are actually valid. The argument must either be the value of the macro EOF (which has a negative value), or must be a non-negative value within the range representable as *unsigned char*. Passing invalid values leads to undefined behavior.

Values of type *int* that were returned by getc(3), fgetc(3), and similar functions or macros are already in the correct range, and may be safely passed to these **ctype** functions without any casts.

Values of type *char* or *signed char* must first be cast to *unsigned char*, to ensure that the values are within the correct range. The result should then be cast to *int* to avoid warnings from some compilers. Casting a negative-valued *char* or *signed char* directly to *int* will produce a negative-valued *int*, which will be outside the range of allowed values (unless it happens to be equal to EOF, but even that would not give the desired result).

NAME

curses — screen functions with “optimal” cursor motion

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

cc [*flags*] *files* **-lcurses** [*libraries*]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the **refresh**() tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine **initscr**() must be called before any of the other routines that deal with windows and screens are used. The routine **endwin**() should be called before exiting. The routine **start_color**() must be called before any of the other routines that deal with color are used.

SEE ALSO

ioctl(2), getenv(3), tty(4), termcap(5)

Ken Arnold, *Screen Updating and Cursor Movement Optimization: A Library Package*.

AUTHORS

Ken Arnold

FUNCTIONS

Function Name	Manual Page Name
addch	curses_addch(3)
addchnstr	curses_addchstr(3)
addchstr	curses_addchstr(3)
addnstr	curses_addstr(3)
addstr	curses_addstr(3)
assume_default_colors	curses_default_colors(3)
attr_get	curses_attributes(3)
attr_off	curses_attributes(3)
attr_on	curses_attributes(3)
attr_set	curses_attributes(3)
attroff	curses_attributes(3)
attron	curses_attributes(3)
attrset	curses_attributes(3)
beep	curses_tty(3)
bkgd	curses_background(3)
bkgdset	curses_background(3)
border	curses_border(3)
box	curses_border(3)
can_change_color	curses_color(3)
cbreak	curses_tty(3)
clear	curses_clear(3)
clearok	curses_clear(3)
clrtobot	curses_clear(3)

clrtoeol	curses_clear(3)
color_content	curses_color(3)
color_set	curses_attributes(3)
copywin	curses_window(3)
curs_set	curses_tty(3)
def_prog_mode	curses_tty(3)
def_shell_mode	curses_tty(3)
define_key	curses_input(3)
delay_output	curses_tty(3)
delch	curses_delch(3)
deleteln	curses_deleteln(3)
delscreen	curses_screen(3)
delwin	curses_window(3)
derwin	curses_window(3)
doupdate	curses_refresh(3)
dupwin	curses_window(3)
echo	curses_tty(3)
endwin	curses_screen(3)
erase	curses_clear(3)
erasechar	curses_tty(3)
flash	curses_tty(3)
flushinp	curses_tty(3)
flushok	curses_refresh(3)
fullname	curses_termcap(3)
getattrs	curses_attributes(3)
getbegx	curses_cursor(3)
getbegy	curses_cursor(3)
getbkgd	curses_background(3)
getcap	curses_termcap(3)
getch	curses_input(3)
getcurx	curses_cursor(3)
getcury	curses_cursor(3)
getmaxx	curses_cursor(3)
getmaxy	curses_cursor(3)
getnstr	curses_input(3)
getparx	curses_cursor(3)
getpary	curses_cursor(3)
getparyx	curses_cursor(3)
getstr	curses_input(3)
gettmode	curses_tty(3)
getwin	curses_fileo(3)
getyx	curses_cursor(3)
has_colors	curses_color(3)
has_ic	curses_tty(3)
has_il	curses_tty(3)
hline	curses_line(3)
idcok	curses_tty(3)
idlok	curses_tty(3)
inch	curses_inch(3)

inchnstr	curses_inch(3)
inchstr	curses_inch(3)
init_color	curses_color(3)
init_pair	curses_color(3)
initscr	curses_screen(3)
innstr	curses_inch(3)
insch	curses_insertch(3)
insdelln	curses_insdelln(3)
insertln	curses_insertln(3)
instr	curses_inch(3)
intrflush	curses_tty(3)
is_linetouched	curses_touch(3)
is_wintouched	curses_touch(3)
isendwin	curses_screen(3)
keyname	curses_keyname(3)
keyok	curses_input(3)
keypad	curses_input(3)
killchar	curses_tty(3)
leaveok	curses_tty(3)
longname	curses_termcap(3)
meta	curses_tty(3)
move	curses_cursor(3)
mvaddch	curses_addch(3)
mvaddchnstr	curses_addchstr(3)
mvaddchstr	curses_addchstr(3)
mvaddnstr	curses_addstr(3)
mvaddstr	curses_addstr(3)
mvcur	curses_cursor(3)
mvderwin	curses_window(3)
mvgetnstr	curses_input(3)
mvgetstr	curses_input(3)
mvhline	curses_line(3)
mvinchstr	curses_inch(3)
mvinchnstr	curses_inch(3)
mvprintw	curses_print(3)
mvscanw	curses_scanw(3)
mvvline	curses_line(3)
mvwaddch	curses_addch(3)
mvwaddchnstr	curses_addchstr(3)
mvwaddchstr	curses_addchstr(3)
mvwaddnstr	curses_addstr(3)
mvwaddstr	curses_addstr(3)
mvwgetnstr	curses_input(3)
mvwgetstr	curses_input(3)
mvwhline	curses_line(3)
mvwinchstr	curses_inch(3)
mvwinchnstr	curses_inch(3)
mvwprintw	curses_print(3)
mvwscanw	curses_scanw(3)

mvwvline	curses_line(3)
napms	curses_tty(3)
newpad	curses_pad(3)
newterm	curses_screen(3)
newwin	curses_window(3)
nl	curses_tty(3)
nocbreak	curses_tty(3)
nodelay	curses_input(3)
noecho	curses_tty(3)
nonl	curses_tty(3)
noqiflush	curses_tty(3)
noraw	curses_tty(3)
notimeout	curses_input(3)
overlay	curses_window(3)
overwrite	curses_window(3)
pair_content	curses_color(3)
pnoutrefresh	curses_pad(3)
prefresh	curses_pad(3)
printw	curses_print(3)
putwin	curses_fileo(3)
qiflush	curses_tty(3)
raw	curses_tty(3)
redrawwin	curses_touch(3)
refresh	curses_refresh(3)
reset_prog_mode	curses_tty(3)
reset_shell_mode	curses_tty(3)
resetty	curses_tty(3)
resizeterm	curses_screen(3)
savetty	curses_tty(3)
scanw	curses_scanw(3)
scrl	curses_scroll(3)
scroll	curses_scroll(3)
scrollok	curses_scroll(3)
set_term	curses_screen(3)
setscrreg	curses_scroll(3)
setterm	curses_screen(3)
standend	curses_standout(3)
standout	curses_standout(3)
start_color	curses_color(3)
subpad	curses_pad(3)
subwin	curses_window(3)
termattrs	curses_attributes(3)
timeout	curses_input(3)
touchline	curses_touch(3)
touchoverlap	curses_touch(3)
touchwin	curses_touch(3)
unctrl	curses_print(3)
underend	curses_underscore(3)
underscore	curses_underscore(3)

ungetch	curses_input(3)
untouchwin	curses_touch(3)
use_default_colors	curses_default_colors(3)
vline	curses_line(3)
waddch	curses_addch(3)
waddchnstr	curses_addchstr(3)
waddchstr	curses_addchstr(3)
waddnstr	curses_addstr(3)
waddstr	curses_addstr(3)
wattr_get	curses_attributes(3)
wattr_off	curses_attributes(3)
wattr_on	curses_attributes(3)
wattr_set	curses_attributes(3)
wattroff	curses_attributes(3)
wattron	curses_attributes(3)
wattrset	curses_attributes(3)
wbkgd	curses_background(3)
wbkgdset	curses_background(3)
wborder	curses_border(3)
wclear	curses_clear(3)
wclrtobot	curses_clear(3)
wclrtoeol	curses_clear(3)
wcolor_set	curses_attributes(3)
wdelch	curses_delch(3)
wdeleteln	curses_deleteln(3)
werase	curses_clear(3)
wgetch	curses_input(3)
wgetnstr	curses_input(3)
wgetstr	curses_input(3)
whline	curses_line(3)
winch	curses_inch(3)
winchnstr	curses_inch(3)
winchstr	curses_inch(3)
winnstr	curses_inch(3)
winsch	curses_insertch(3)
winsdelln	curses_insdelln(3)
winsertln	curses_insertln(3)
winstr	curses_inch(3)
wmove	curses_cursor(3)
wnoutrefresh	curses_refresh(3)
wprintw	curses_print(3)
wredrawln	curses_touch(3)
wrefresh	curses_refresh(3)
wresize	curses_window(3)
wscanw	curses_scanw(3)
wscrl	curses_scroll(3)
wsetscrreg	curses_scroll(3)
wstandend	curses_standout(3)
wstandout	curses_standout(3)

wtimeout	curses_input(3)
wtouchln	curses_touch(3)
wunderend	curses_underscore(3)
wunderscore	curses_underscore(3)
wvline	curses_line(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_addch, **addch**, **waddch**, **mvaddch**, **mvwaddch** — curses add characters to windows routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
addch(chtype ch);

int
waddch(WINDOW *win, chtype ch);

int
mvaddch(int y, int x, chtype ch);

int
mvwaddch(WINDOW *win, int y, int x, chtype ch);
```

DESCRIPTION

These functions add characters to `stdscr` or to the specified window.

The **addch**() function adds the character given in *ch* to `stdscr` at the current cursor position and advances the current cursor position by one. Any character attributes set in *ch* will be merged with the background attributes currently set on `stdscr`.

The **waddch**() function is the same as the **addch**() function, excepting that the character is added to the window specified by *win*.

The **mvaddch**() and **mvwaddch**() functions are the same as the **addch**() and **waddch**() functions, respectively, excepting that **wmove**() is called to move the cursor to the position specified by *y*, *x* before the character is added to the window.

LINE DRAWING CHARACTERS

Some terminals support the display of line drawing and graphics characters. These characters can be added using their defined names, as shown in the table below. Where the terminal does not support a specific character, the default (non-graphics) character is displayed instead.

Name	Default	Description
ACS_RARROW	>	Arrow pointing right
ACS_LARROW	<	Arrow pointing left
ACS_UARROW	^	Arrow pointing up
ACS_DARROW	v	Arrow pointing down
ACS_BLOCK	#	Solid square block
ACS_DIAMOND	+	Diamond
ACS_CKBOARD	:	Checker board (stipple)
ACS_DEGREE	'	Degree symbol
ACS_PLMINUS	#	Plus/minus
ACS_BOARD	#	Board of squares
ACS_LANTERN	#	Lantern symbol
ACS_LRCORNER	+	Lower right-hand corner

ACS_URCORNER		+Upper right-hand corner
ACS_ULCORNER		+Upper left-hand corner
ACS_LLCORNER	+	Lower left-hand corner
ACS_PLUS	+	Plus
ACS_HLINE	-	Horizontal line
ACS_S1	-	Scan line 1
ACS_S9	-	Scan line 9
ACS_LTEE	+	Left tee
ACS_RTEE	+	Right tee
ACS_BTEE	+	Bottom tee
ACS_TTEE	+	Top tee
ACS_VLINE	|	Vertical line
ACS_BULLET	o	Bullet

The following additional, *ncurses* compatible, characters are also supported.

Name	**Default**	**Description**
ACS_S3	-	Scan line 3
ACS_S7	-	Scan line 7
ACS_LEQUAL	<	Less than or equal to
ACS_GEQUAL	>	Greater than or equal to
ACS_PI	*	Pi symbol
ACS_NEQUAL	!	Not equal to
ACS_STERLING	f	Sterling symbol

For compatibility with some *System V* implementations, the following definitions are also supported.

System V Name	**NetBSD Curses Name**
ACS_SBBS	ACS_LRCORNER
ACS_BBSS	ACS_URCORNER
ACS_BSSB	ACS_ULCORNER
ACS_SSBB	ACS_LLCORNER
ACS_SSSS	ACS_PLUS
ACS_BSBS	ACS_HLINE
ACS_SSSB	ACS_LTEE
ACS_SBSS	ACS_RTEE
ACS_SSBS	ACS_BTEE
ACS_BSSS	ACS_TTEE
ACS_SBSB	ACS_VLINE

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_addchstr`(3), `curses_addstr`(3), `curses_attributes`(3), `curses_cursor`(3), `curses_delch`(3), `curses_inch`(3), `curses_insertch`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_addchstr, **addchstr**, **waddchstr**, **addchnstr**, **waddchnstr**, **mvaddchstr**, **mvwaddchstr**, **mvaddchnstr**, **mvwaddchnstr** — curses add character strings to windows routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
addchstr(const chtype *chstr);

int
waddchstr(WINDOW *win, const chtype *chstr);

int
mvaddchstr(int y, int x, const chtype *chstr);

int
mvwaddchstr(WINDOW *win, int y, int x, const chtype *chstr);

int
addchnstr(const chtype *chstr, int n);

int
waddchnstr(WINDOW *win, const chtype *chstr, int n);

int
mvaddchnstr(int y, int x, const chtype *chstr, int n);

int
mvwaddchnstr(WINDOW *win, int y, int x, const chtype *chstr, int n);
```

DESCRIPTION

These functions add character strings and attributes to `stdscr` or to the specified window.

The **addchstr**() function will add the characters and their attributes passed in *chstr* to `stdscr` starting at the current cursor position. Any character attributes set in *chstr* will be merged with the background attributes currently set on `stdscr`. The **waddstr**() function does the same as **addchstr**() but adds the string to the window specified by **win**().

The **addchnstr**() function will add the contents of *string* to `stdscr` but will limit the number of characters added to be, at most, *n*. If *n* is −1 then *addchnstr* will add the number of characters contained in the null terminated string *chstr*. Any character attributes set in *chstr* will be merged with the background attributes currently set on `stdscr`.

The **waddchnstr**() function does the same as *addchnstr* but adds the string to the window specified by *win*.

The functions **mvaddchstr**(), **mwaddchnstr**(), **mvwaddchstr**() and **mvwaddchnstr**() are the same as the functions **addchstr**(), **waddchstr**(), **waddchstr**() and **waddchnstr**(), respectively, except that **wmove**() is called to move the cursor to the position specified by *y*, *x* before the string is added to the window.

RETURN VALUES

The functions will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_addch`(3), `curses_addstr`(3), `curses_attributes`(3), `curses_cursor`(3), `curses_inch`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

These functions first appeared in NetBSD 2.0.

NAME

curses_addstr, **addstr**, **waddstr**, **addnstr**, **waddnstr**, **mvaddstr**, **mvwaddstr**, **mvaddnstr**, **mvwaddnstr** — curses add character strings to windows routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
addstr(const char *string);

int
waddstr(WINDOW *win, const char *string);

int
mvaddstr(int y, int x, const char *string);

int
mvwaddstr(WINDOW *win, int y, int x, const char *string);

int
addnstr(const char *string, int len);

int
waddnstr(WINDOW *win, const char *string, int len);

int
mvaddnstr(int y, int x, const char *string, int len);

int
mvwaddnstr(WINDOW *win, int y, int x, const char *string, int len);
```

DESCRIPTION

These functions add character strings to `stdscr` or to the specified window.

The **addstr**() function will add the characters passed in *string* to `stdscr` starting at the current cursor position. Any background attributes currently set on `stdscr` will be applied to the added character. The **waddstr**() function does the same as **addstr**() but adds the string to the window specified by **win**().

The **addnstr**() function will add the contents of *string* to `stdscr` but will limit the number of characters added to be, at most, *len*. If *len* is −1 then *addnstr* will add the number of characters contained in the null terminated string *string*. Any background attributes currently set on `stdscr` will be applied to the added character. The **waddnstr**() function does the same as *addnstr* but adds the string to the window specified by *win*.

The functions **mvaddstr**(), **mwaddnstr**(), **mvwaddstr**() and **mvwaddnstr**() are the same as the functions **addstr**(), **waddstr**(), **waddstr**() and **waddnstr**(), respectively, excepting that **wmove**() is called to move the cursor to the position specified by *y*, *x* before the string is added to the window.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

OK The function completed successfully.
ERR An error occurred in the function.

SEE ALSO

curses_addch(3), curses_addchstr(3), curses_attributes(3), curses_cursor(3), curses_inch(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_attributes, **attron**, **attroff**, **attrset**, **color_set**, **getattrs**, **termattrs**, **wattron**, **wattroff**, **wattrset**, **wcolor_set**, **attr_on**, **attr_off**, **attr_set**, **attr_get**, **term_attrs**, **wattr_on**, **wattr_off**, **wattr_set**, **wattr_get** — curses general attribute manipulation routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
attron(int attr);

int
attroff(int attr);

int
attrset(int attr);

int
color_set(short pair, void *opt);

chtype
getattrs(WINDOW *win);

chtype
termattrs(void);

int
wcolor_set(WINDOW *win, short pair, void *opt);

int
wattron(WINDOW * win, int attr);

int
wattroff(WINDOW * win, int attr);

int
wattrset(WINDOW * win, int attr);

int
attr_on(attr_t attr, void *opt);

int
attr_off(attr_t attr, void *opt);

int
attr_set(attr_t attr, short pair, void *opt);

int
attr_get(attr_t *attr, short *pair, void *opt);

attr_t
term_attrs(void);

int
wattr_on(WINDOW *win, attr_t attr, void *opt);
```

int
wattr_off(*WINDOW *win*, *attr_t attr*, *void *opt*);

int
wattr_set(*WINDOW *win*, *attr_t attr*, *short pair*, *void *opt*);

int
wattr_get(*WINDOW *win*, *attr_t *attr*, *short *pair*, *void *opt*);

DESCRIPTION

These functions manipulate attributes on `stdscr` or on the specified window. The attributes that can be manipulated are:

A_NORMAL	no special attributes are applied
A_STANDOUT	characters are displayed in standout mode
A_UNDERLINE	characters are displayed underlined
A_REVERSE	characters are displayed in inverse video
A_BLINK	characters blink
A_DIM	characters are displayed at a lower intensity
A_BOLD	characters are displayed at a higher intensity
A_INVIS	characters are added invisibly
A_PROTECT	characters are protected from modification
A_ALTCHARSET	characters are displayed using the alternate character set (ACS)
COLOR_PAIR(n)	characters are displayed using color pair n.

The **attron**() function turns on the attributes specified in *attr* on `stdscr`, while the **attroff**() function turns off the attributes specified in *attr* on `stdscr`.

The function **attrset**() sets the attributes of `stdscr` to those specified in *attr*, turning off any others. To turn off all the attributes (including color and alternate character set), use **attrset**(*A_NORMAL*).

Multiple attributes can be manipulated by combining the attributes using a logical *OR*. For example, **attron**(*A_REVERSE | A_BOLD*) will turn on both inverse video and higher intensity.

The function **color_set**() sets the color pair attribute to the pair specified in *pair*.

The function **getattrs**() returns the attributes that are currently applied to window specified by *win*.

The function **termattrs**() returns the logical *OR* of attributes that can be applied to the screen.

The functions **wattron**(), **wattroff**(), **wattrset**(), and **wcolor_set**() are equivalent to **attron**(), **attroff**() **attrset**(), and **color_set**() respectively, excepting that the attributes are applied to the window specified by *win*.

The following functions additionally manipulate wide attributes on `stdscr` or on the specified window. The additional wide attributes that can be manipulated are:

WA_STANDOUT	characters are displayed in standout mode
WA_UNDERLINE	characters are displayed underlined
WA_REVERSE	characters are displayed in inverse video
WA_BLINK	characters blink
WA_DIM	characters are displayed at a lower intensity
WA_BOLD	characters are displayed at a higher intensity
WA_INVIS	characters are added invisibly
WA_PROTECT	characters are protected from modification

WA_ALTCHARSET	characters are displayed using the alternate character set (ACS)
WA_LOW	characters are displayed with low highlight
WA_TOP	characters are displayed with top highlight
WA_HORIZONTAL	characters are displayed with horizontal highlight
WA_VERTICAL	characters are displayed with vertical highlight
WA_LEFT	characters are displayed with left highlight
WA_RIGHT	characters are displayed with right highlight

The **attr_on**() function turns on the wide attributes specified in *attr* on stdscr, while the **attr_off**() function turns off the wide attributes specified in *attr* on stdscr.

The function **attr_set**() sets the wide attributes of stdscr to those specified in *attr* and *pair*, turning off any others. Note that a color pair specified in *pair* will override any color pair specified in *attr*.

The function **attr_get**() sets *attr* to the wide attributes and *pair* to the color pair currently applied to stdscr. Either of *attr* and *pair* can be NULL, if the relevant value is of no interest.

The function **term_attrs**() returns the logical *OR* of wide attributes that can be applied to the screen.

The functions **wattr_on**(), **wattr_off**() and **wattr_set**() are equivalent to **attr_on**(), **attr_off**() and **attr_set**() respectively, excepting that the character is added to the window specified by *win*.

The function **wattr_get**() is equivalent to **attr_get**(), excepting that the wide attributes and color pair currently applied to *win* are set.

The following constants can be used to extract the components of a chtype:

A_ATTRIBUTES	bit-mask containing attributes part
A_CHARTEXT	bit-mask containing character part
A_COLOR	bit-mask containing color-pair part

RETURN VALUES

These functions return OK on success and ERR on failure.

SEE ALSO

curses_addch(3), curses_addchstr(3), curses_addstr(3), curses_background(3), curses_color(3), curses_insertch(3), curses_standout(3), curses_underscore(3)

NOTES

The *opt* argument is not currently used but is reserved for a future version of the specification.

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

The **getattrs**() function is a NetBSD extension.

HISTORY

These functions first appeared in NetBSD 1.5.

BUGS

Some terminals do not support characters with both color and other attributes set. In this case, the other attribute is displayed instead of the color attribute.

NAME

curses_background, **bkgd**, **bkgdset**, **getbkgd**, **wbkgd**, **wbkgdset** — curses attribute manipulation routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
bkgd(chtype);

int
bkgdset(chtype);

chtype
getbkgd(WINDOW *);

int
wbkgd(chtype);

int
wbkgdset(chtype);
```

DESCRIPTION

These functions manipulate the background attributes on `stdscr` or on the specified window.

The function **wbkgdset**(*win*, *ch*) sets the background attributes of the specified window *win* to *ch*.

When the background attributes are set on a window, characters are added to the window with the logical *OR* of the background attributes and the character's attributes. If both the background attribute and the character attribute contain color, the color of the character attribute is rendered. If the background attribute contains a non-space character, then this character is added where the foreground character is a space character.

Note that subwindows created from *win* inherit the background attributes of *win*.

The function **wbkgd**(*win*, *ch*) sets the background attributes of the specified window *win* to *ch* and also sets the rendition of every character position on that window, as if the characters had been newly added to *win*. The rendition of characters on subwindows of *win* is also set to *ch*.

The functions **bkgdset**(*ch*) and **bkgd**(*ch*) are equivalent to **wbkgdset**(*stdscr*, *ch*) and **wbkgd**(*stdscr*, *ch*), respectively.

The function **getbkgd**(*win*) returns the background attributes for the window *win*.

RETURN VALUES

The functions **wbkgdset**() and **wbkgd**() return OK on success and ERR on failure.

SEE ALSO

`curses_attributes`(3), `curses_color`(3), `curses_window`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

These functions first appeared in NetBSD 1.6.

NAME

curses_border, **border**, **box**, **wborder** — curses border drawing routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
border(chtype ls, chtype rs, chtype ts, chtype bs, chtype tl, chtype tr,
    chtype bl, chtype br);

int
box(WINDOW *win, chtype vertical, chtype horizontal);

int
wborder(WINDOW *win, chtype ls, chtype rs, chtype ts, chtype bs, chtype tl,
    chtype tr, chtype bl, chtype br);
```

DESCRIPTION

These functions draw borders around `stdscr` or around the specified window.

The **border**() function draws a border around `stdscr` using the characters given as arguments to the function. The *ls*, *rs*, *ts* and *bs* are the characters used to draw the left, right, top and bottom sides, respectively. The *tl*, *tr*, *bl* and *br* are the characters used to draw the top-left, top-right, bottom-left and bottom-right corners, respectively. If any of the characters have a text portion that is 0 then a default alternate character set character is used for that character. Note that even though the text portion of the argument is 0, the argument can still be used to specify the attributes for that portion of the border. The following table shows the default characters for each argument:

ls	ACS_VLINE
rs	ACS_VLINE
ts	ACS_HLINE
bs	ACS_HLINE
tl	ACS_ULCORNER
tr	ACS_URCORNER
bl	ACS_LLCORNER
br	ACS_LRCORNER

wborder() is the same as **border**() excepting that the border is drawn around the specified window.

The **box**() command draws a box around the window given in *win* using the *vertical* character for the vertical lines and the *horizontal* character for the horizontal lines. The corner characters of this box will be the defaults as described for **border**() above. Passing characters with text portion that is 0 to **box**() will result in the same defaults as those for **border**() as described above.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK`	The function completed successfully.
`ERR`	An error occurred in the function.

SEE ALSO

`curses_attributes`(3), `curses_line`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_clear, **clear**, **wclear**, **clearok**, **clrtobot**, **clrtoeol**, **erase**, **werase**, **wclrtobot**, **wclrtoeol** — curses clear window routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
clear(void);

int
clearok(WINDOW *win, bool flag);

int
clrtobot(void);

int
clrtoeol(void);

int
erase(void);

int
wclear(WINDOW *win);

int
werase(WINDOW *win);

int
wclrtobot(WINDOW *win);

int
wclrtoeol(WINDOW *win);
```

DESCRIPTION

These functions clear all or part of `stdscr` or of the specified window.

The **clear**() and **erase**() functions erase all characters on `stdscr`. **wclear**() and **werase**() perform the same function as **clear**() and **erase**(), respectively, excepting that the specified window is cleared.

By setting *flag* to `TRUE`, the **clearok**() function is used to force the next call to **wrefresh**() to clear and completely redraw the window given in *win*.

The function **clrtobot**() will clear `stdscr` from the current row to the bottom of the screen. **clrtoeol**() will clear `stdscr` from the current column position to the end of the line. **wclrtobot**() and **wclrtoeol**() are the same as **clrtobot**() and **clrtoeol**(), respectively, excepting the window specified is operated on instead of `stdscr`.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.

ERR An error occurred in the function.

SEE ALSO

curses_refresh(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_color, **has_colors**, **can_change_color**, **start_color**, **init_pair**, **pair_content**, **COLOR_PAIR**, **PAIR_NUMBER**, **init_color**, **color_content**, **no_color_video** — curses color manipulation routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

bool
has_colors(void);

bool
can_change_color(void);

int
start_color(void);

int
init_pair(short pair, short fore, short back);

int
pair_content(short pair, short *fore, short *back);

int
COLOR_PAIR(int n);

int
PAIR_NUMBER(int val);

int
init_color(short color, short red, short green, short blue);

int
color_content(short color, short *red, short *green, short *blue);

attr_t
no_color_video(void);

extern int COLOR_PAIRS;

extern int COLORS;
```

DESCRIPTION

These functions manipulate color on terminals that support color attributes.

The function **has_colors**() indicates whether a terminal is capable of displaying color attributes. It returns TRUE if the terminal is capable of displaying color attributes and FALSE otherwise.

The function **can_change_color**() indicates whether a terminal is capable of redefining colors. It returns TRUE if colors can be redefined and FALSE if they can not.

The function **start_color**() initializes the curses color support on a terminal. It must be called before any color manipulation functions are called on that terminal. The function initializes the eight basic colors (black, red, green, yellow, blue, magenta, cyan and white) that are specified using the color macros (such as COLOR_BLACK) defined in <*curses.h*>. **start_color**() also initializes the global external variables *COLORS* and *COLOR_PAIRS*. *COLORS* defines the number of colors that the terminal supports and

COLOR_PAIRS defines the number of color-pairs that the terminal supports. These color-pairs are initialized to white foreground on black background. **start_color**() sets the colors on the terminal to the curses defaults of white foreground on black background unless the functions **assume_default_colors**() or **use_default_colors**() have been called previously.

The function **init_pair**(*pair*, *fore*, *back*) sets foreground color *fore* and background color *back* for color-pair number *pair*. The valid range for the color-pair *pair* is from 1 to *COLOR_PAIRS* - 1 and the valid range for the colors is any number less than *COLORS*. Specifying a negative number will set that color to the default foreground or background color. The 8 initial colors are defined as:

COLOR_BLACK
COLOR_RED
COLOR_GREEN
COLOR_YELLOW
COLOR_BLUE
COLOR_MAGENTA
COLOR_CYAN
COLOR_WHITE

Color-pair 0 is used as the default color pair, so changing this will have no effect. Use the function **assume_default_colors**() to change the default colors.

The function **pair_content**(*pair*, **fore*, **back*) stores the foreground and background color numbers of color-pair *pair* in the variables *fore* and *back*, respectively.

The macro **COLOR_PAIR**(*n*) gives the attribute value of color-pair number *n*. This is the value that is used to set the attribute of a character to this color-pair. For example,

```
attrset(COLOR_PAIR(2))
```

will display characters using color-pair 2.

The macro **PAIR_NUMBER**(*val*) gives the color-pair number associated with the attribute value *val*.

The function **init_color**(*color*, *red*, *green*, *blue*) sets the red, green and blue intensity components of color *color* to the values *red*, *green* and *blue*, respectively. The minimum intensity value is 0 and the maximum intensity value is 1000.

The function **color_content**(*color*, **red*, **green*, **blue*) stores the red, green and blue intensity components of color *color* in the variables *red*, *green*, and *blue*, respectively.

The function **no_color_video**() returns those attributes that a terminal is unable to combine with color.

RETURN VALUES

The functions **start_color**(), **init_pair**(), **pair_content**(), **init_color**() and **color_content**() return OK on success and ERR on failure.

SEE ALSO

curses_attributes(3), curses_background(3), curses_default_colors(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

The function **no_color_video**() and the use of negative color numbers are extensions to the X/Open Curses specification.

HISTORY

These functions first appeared in NetBSD 1.5.

NAME

curses_cursor, **getcury**, **getcurx**, **getyx**, **getbegy**, **getbegx**, **getbegyx**, **getmaxy**, **getmaxx**, **getmaxyx**, **getpary**, **getparx**, **getparyx**, **move**, **wmove**, **mvcur** — curses cursor and window location and positioning routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
getcury(WINDOW *win);

int
getcurx(WINDOW *win);

void
getyx(WINDOW *win, int y, int x);

int
getbegy(WINDOW *win);

int
getbegx(WINDOW *win);

void
getbegyx(WINDOW *win, int y, int x);

int
getmaxy(WINDOW *win);

int
getmaxx(WINDOW *win);

void
getmaxyx(WINDOW *win, int y, int x);

int
getpary(WINDOW *win);

int
getparx(WINDOW *win);

void
getparyx(WINDOW *win, int y, int x);

int
move(int y, int x);

int
wmove(WINDOW *win, int y, int x);

int
mvcur(int oldy, int oldx, int y, int x);
```

DESCRIPTION

These functions and macros locate and position cursors and windows.

The **getcury**() and **getcurx**() functions get the current row and column positions, respectively, of the cursor in the window *win*. The **getyx**() macro sets the values of *y* and *x* to the current row and column positions of the cursor in the window *win*.

The origin row and columns of a window *win* can be determined by calling the **getbegy**() and **getbegx**() functions, respectively, and the maximum row and column for the window can be found by calling the functions **getmaxy**() and **getmaxx**(), respectively. The **getbegyx**() and **getmaxyx**() macros set the values of *y* and *x* to the origin and maximum row and column positions, respectively, for the window *win*.

The **getpary**() and **getparx**() functions return the row and column position of the given subwindow relative to the window's parent. The macro **getparyx**() sets the values of *y* and *x* to the origin of the subwindow relative to the window's parent.

The **move**() function positions the cursor on the current window at the position given by *y*, *x*. The cursor position is not changed on the screen until the next **refresh**().

The **wmove**() function is the same as the **move**() function, excepting that the cursor is moved in the window specified by *win*.

The function **mvcur**() moves the cursor to *y*, *x* on the screen. The arguments *oldy*, *oldx* define the previous cursor position for terminals that do not support absolute cursor motions. The curses library may optimise the cursor motion based on these values. If the **mvcur**() succeeds then the curses internal structures are updated with the new position of the cursor. If the destination arguments for **mvcur**() exceed the terminal bounds an error will be returned and the cursor position will be unchanged.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following values:

OK The function completed successfully.
ERR An error occurred in the function.

SEE ALSO

curses_refresh(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification. The **getbegx**(), **getbegy**(), **getcurx**(), **getcury**(), **getmaxx**(), **getmaxy**(), **getparx**(), and **getpary**() functions are extensions.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_default_colors, **assume_default_colors**, **use_default_colors** — curses default colors setting routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
assume_default_colors(short fore, short back);

int
use_default_colors();
```

DESCRIPTION

These functions tell the curses library to set the default colors or to use the terminal's default colors instead of using the default colors for curses applications (which are white foreground on black background).

The function **assume_default_colors**(*fore*, *back*) sets the default colors to foreground color *fore* and background color *back*. If a value of −1 is used for a color, then the terminal default color is used for that color.

The function **use_default_colors**() sets both the foreground and background colors to the terminal default colors. This is equivalent to **assume_default_colors**(*−1*, *−1*).

RETURN VALUES

These functions return OK on success and ERR on failure.

SEE ALSO

curses_color(3)

STANDARDS

These functions are based on *ncurses* extensions to the curses standards.

HISTORY

These functions first appeared in NetBSD 2.0.

NAME

curses_delch, **delch**, **wdelch** — curses delete characters routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
delch(void);

int
wdelch(WINDOW *win);
```

DESCRIPTION

These functions delete characters from `stdscr` or from the specified window.

The **delch**() function deletes the character at the current cursor position on `stdscr`. Characters to the right of the deleted character are moved one position to the left. The cursor position is unchanged.

The **wdelch**() function is the same as the **delch**() function, excepting that the character is deleted from the specified window.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_addch`(3), `curses_insertch`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_deleteln, **deleteln**, **wdeleteln** — curses delete single line routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
deleteln(void);

int
wdeleteln(WINDOW *win);
```

DESCRIPTION

These functions delete a single line from `stdscr` or from the specified window.

The **deleteln**() function deletes the screen line containing the cursor in the `stdscr`. The **wdeleteln**() function is the same as the **deleteln**() function, excepting that the line is deleted from the window specified by *win*.

All lines following the deleted line are moved up one line toward the cursor. The last line of the window is cleared. The cursor position is unchanged.

If a scrolling region has been set with the **setscrreg**() or **wsetscrreg**() functions and the current cursor position is inside the scrolling region, then only the lines from the current line to the bottom of the scrolling region are moved up and the bottom line of the scrolling region cleared.

The functions **deleteln**() and **wdeleteln**(*win*) are equivalent to **winsdelln**(*stdscr*, *−1*) and **winsdelln**(*win*, *−1*) respectively.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_insdelln`(3), `curses_insertln`(3), `curses_scroll`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_echochar, **echochar**, **wechochar**, **pechochar** — curses add characters and then refresh routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
echochar(const chtype ch);

int
wechochar(WINDOW *win, const chtype ch);

int
pechochar(WINDOW *pad, const chtype ch);
```

DESCRIPTION

These functions add characters to `stdscr` or to the specified window or pad and then cause an immediate **refresh**() of that window or pad.

The **echochar**() function adds the character given in *ch* to `stdscr` at the current cursor position and advances the current cursor position by one. Any character attributes set in *ch* will be merged with the background attributes currently set on `stdscr`. `stdscr` is then refreshed. Calling **echochar**() is equivalent to calling **addch**() followed by **refresh**().

The **wechochar**() function is the same as the **echochar**() function, excepting that the character is added to the window specified by *win* and *win* is refreshed.

The **pechochar**() function is the similar to the **echochar**() function, excepting that the character is added to the pad specified by *pad* and *pad* is refreshed at its previous location on the screen. Calling **pechochar**() is equivalent to calling **addch**() followed by **prefresh**().

RETURN VALUES

These functions will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_addch`(3), `curses_attributes`(3), `curses_pad`(3), `curses_refresh`(3)

STANDARDS

The **echochar**(), **wechochar**(), and **pechochar**() functions comply with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_fileio, **getwin**, **putwin** — curses file input/output routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

WINDOW *
getwin(FILE *fp);

int
putwin(WINDOW *win, FILE *fp);
```

DESCRIPTION

These functions read and write data to and from files.

The **getwin**() function reads window data that has been stored in the file to which *fp* points, and then creates a new window using that data.

The **putwin**() function writes window data from the window *win* to the file pointed to by *fp*.

RETURN VALUES

The **getwin**() function returns one of the following values:

OK The function completed successfully.
ERR An error occurred in the function.

The **putwin**() function returns NULL if an error is detected.

SEE ALSO

curses_window(3), fread(3,) fwrite(3)

NOTES

Subwindows can not be created by the **getwin**() function, nor written by the **putwin**() function.

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

These functions first appeared in NetBSD 5.0.

NAME

curses_inch, **inch**, **winch**, **inchnstr**, **mvinchnstr**, **winchnstr**, **mvwinchnstr**, **inchstr**, **mvinchstr**, **winchstr**, **mvwinchstr**, **innstr**, **winnstr**, **mvinnstr**, **mvwinnstr**, **instr**, **winstr** **mvinstr**, **mvwinstr** — curses read screen contents routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

chtype
inch(void);

chtype
winch(WINDOW *win);

int
inchnstr(chtype *chars, int n);

int
mvinchnstr(int y, int x, chtype *chstr, int n);

int
winchnstr(WINDOW *win, chtype *chars, int n);

int
mvwinchnstr(WINDOW *win, int y, int x, chtype *chstr, int n);

int
inchstr(chtype *chars);

int
mvinchstr(int y, int x, chtype *chstr);

int
winchstr(WINDOW *win, chtype *chars);

int

mvwinchstr WINDOW *win int y int x chtype *chstr

int
innstr(char *str, int n);

int
winnstr(WINDOW *win, char *str, int n);

int
mvinnstr(int y, int x, char *str, int n);

int
mvwinnstr(WINDOW *win, int y, int x, char *str, int n);

int
instr(char *str);

int
winstr(WINDOW *win, char *str);
```

int
mvinstr(*int y*, *int x*, *char *str*);

int
mvwinstr(*WINDOW *win*, *int y*, *int x*, *char *str*);

DESCRIPTION

These functions read the contents of `stdscr` or of the specified window.

The **inch**() function returns the character that is displayed on `stdscr` at the current cursor position.

The **winch**() function is the same as the **inch**() function, excepting that the character is read from window specified by *win*.

The **inchnstr**() function fills an array of *chtype* with characters read from `stdscr`, the characters are read starting from the current cursor position and continuing until either n − 1 characters are read or the right hand side of the screen is reached. The resulting character array will be `NULL` terminated.

The **winchnstr**() function is the same as **inchnstr**() excepting that the characters are read from the window specified by *win*.

The **inchstr**() and **winchstr**() functions are the same as the **inchnstr**() and **winchnstr**() functions, respectively, excepting that they do not limit the number of characters read. The characters returned are those from the current starting position to the right hand side of the screen. The use of **inchstr**() and **winchstr**() is not recommended as the character buffer can be overflowed.

The **innstr**() function is similar to the **inchstr**() function, excepting that the array of characters returned is stripped of all the curses attributes making it a plain character string.

The **mvinchstr**(), **mvinchnstr**(), **mvwinchstr**(), and **mvwinchnstr**() functions are the same as the **inchstr**(), **inchnstr**(), **winchstr**(), and **winchstr**() functions, respectively, except that **wmove**() is called to move the cursor to the position specified by *y*, *x* before the output is printed on the window. Likewise, the **mvinstr**(), **mvinnstr**(), **mvwinstr**(), and **mvwinnstr**() functions are the same as the **instr**(), **innstr**(), **winstr**(), and **winstr**() functions, respectively, except that **wmove**() is called to move the cursor to the position specified by *y*, *x* before the output is printed on the window.

The **winnstr**() function is the same as the **innstr**() function, excepting that characters are read from the window specified by *win*.

The **instr**() and **winstr**() functions are the same as the **innstr**() and **winnstr**() functions, respectively, excepting that there are no limits placed on the size of the returned string, which may cause buffer overflows. For this reason, the use of **instr**() and **winstr**() is not recommended.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_addch`(3), `curses_addstr`(3), `curses_attributes`(3), `curses_insertch`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

NOTES

The **inchnstr**() and **innstr**() function read at most n − 1 characters from the screen so as to leave room for NULL termination. The X/Open specification is unclear as to whether or not this is the correct behaviour.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_input, **getch**, **wgetch**, **mvgetch**, **mvwgetch**, **define_key**, **keyok**, **getnstr**, **wgetnstr**, **mvgetnstr**, **mvwgetnstr**, **getstr**, **wgetstr**, **mvgetstr**, **mvwgetstr**, **keypad**, **notimeout**, **timeout**, **wtimeout**, **nodelay**, **ungetch** — curses input stream routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
getch(void);

int
wgetch(WINDOW *win);

int
mvgetch(int y, int x);

int
mvwgetch(WINDOW *win, int y, int x);

int
keyok(int key_symbol, bool flag);

int
define_key(char *sequence, int key_symbol);

int
getnstr(char *str, int limit);

int
wgetnstr(WINDOW *win, char *str, int limit);

int
mvgetnstr(int y, int x, char *str, int limit);

int
mvwgetnstr(WINDOW *win, int y, int x, char *str, int limit);

int
getstr(char *str);

int
wgetstr(WINDOW *win, char *str);

int
mvgetstr(int y, int x, char *str);

int
mvwgetstr(WINDOW *win, int y, int x, char *str);

int
keypad(WINDOW *win, boolf flag);

int
notimeout(WINDOW *win, boolf flag);
```

int
timeout(*int delay*);

int
wtimeout(*WINDOW *win*, *int delay*);

int
nodelay(*WINDOW *win*, *boolf flag*);

int
ungetch(*int c*);

extern int ESCDELAY;

DESCRIPTION

These functions read characters and strings from the window input file descriptor.

The **getch**() function reads a character from the stdscr input file descriptor and returns it. If the **keypad**() flag has been set to TRUE, then **getch**() will assemble multi-character key sequences into key symbols, If the terminal is resized, **getch**() will return KEY_RESIZE, regardlesss of the setting of **keypad**(). Calling **getch**() will cause an implicit **refresh**() on stdscr.

The **wgetch**() function is the same as the **getch**() function, excepting that it reads from the input file descriptor associated with the window specified by *win*.

If the **keypad**() flag is TRUE then the assembly of specific key symbols can be disabled by using the **keyok**() function. If the *flag* is set to FALSE on a key symbol then **getch**() will behave as if the character sequence associated with that key symbol was not recognised and will return the component characters one at a time to the caller.

Custom associations between sequences of characters and a key symbol can be made by using the **define_key**() function. Normally, these associations are made by the information in the termcap(5) database but the **define_key**() function gives the capability to remove or add more associations. If **define_key**() is passed a non-NULL string in *sequence* it will associate that sequence with the key symbol passed in *key_symbol*. The key symbol may be one of the ones listed below or a custom value that is application defined. It is valid to have multiple character sequences map to the same key symbol and there are no constraints on the length of the sequence allowed. The assembly of custom sequences follow the same rules for inter-character timing and so forth as the termcap(5) derived ones. If **define_key**() is passed a NULL in *sequence* then all associations for the key symbol in *key_symbol* will be deleted, this includes any associations that were derived from termcap(5).

The **mvgetch**() and **mvwgetch**() functions are the same as the **getch**() and **wgetch**() functions, respectively, excepting that **wmove**() is called to move the cursor to the position specified by *y*, *x* before the character is read.

Calling **getnstr**(), **wgetnstr**(), **mvgetnstr**() or **mvwgetnstr**() is effectively the same as calling **getch**() repeatedly until a newline is received or the character limit *limit* is reached. Once this happens the string is NULL terminated and returned in *str*. During input, the normal curses input key processing is performed and affects the input buffer. The **mvgetnstr**() function calls **wmove**() to move the cursor to the position given by *y*, *x* before getting the string, **wgetnstr**() reads the input from the designated window, **mvwgetnstr**() moves the cursor to the position given by *y*, *x* before getting the input from the designated window.

The functions **getstr**(), **wgetstr**(), **mvgetstr**(), and **mvwgetstr**() are similar to **getnstr**(), **wgetnstr**(), **mvgetnstr**(), and **mvwgetnstr**(), respectively, excepting that there is no limit on the number of characters that may be inserted into *str*. This may cause the buffer to be overflowed, so their use is not recommended.

The **keypad**() function is used to affect how **getch**() processes input characters. If *flag* is set to TRUE, then **getch**() will scan the input stream looking for multi-character key sequences that are emitted by some terminal function keys. If a recognised sequence of characters is found, then **getch**() will collapse that sequence into an integer key symbol, as shown below. The default setting for the flag is FALSE.

The **notimeout**() function controls whether or not **getch**() will wait indefinitely between characters in a multi-character key sequence or not. If *flag* is TRUE, then there is no timeout applied between characters comprising a multi-character key sequence. If *flag* is FALSE, then the component characters of a multi-character sequence must not have an inter-character gap of more than *ESCDELAY*. If this timing is exceeded, then the multi-character key assembly is deemed to have failed and the characters read thus far are returned one at a time when **getch**() is called. The default setting for the flag is FALSE. The default value of *ESCDELAY* is 300ms. If *ESCDELAY* is negative, no timeout is applied between characters comprising a multi-character key sequence.

The **timeout**() function affects the behaviour of **getch**() when reading a character from stdscr. If *delay* is negative, then **getch**() will block indefinitely on a read. If *delay* is 0, then **getch**() will return immediately with ERR if there are no characters immediately available. If *delay* is a positive number, then **getch**() will wait for that many milliseconds before returning and, if no character was available, then ERR will be returned. Note that for a positive number, the timeout is only accurate to the nearest tenth of a second. Also, the maximum value of *delay* is 25500 milliseconds. The **wtimeout**() function does the same as **timeout**() but applies to the specified window *win*.

The **nodelay**() function turns on and off blocking reads for **getch**(). If *flag* is TRUE, then **getch**() will not block on reads, if *flag* is FALSE, then reads will block. The default setting for the flag is FALSE. **nodelay**(*win*, *TRUE*) is equivalent to **wtimeout**(*win*, *0*) and **nodelay**(*win*, *FALSE*) is equivalent to **wtimeout**(*win*, *-1*).

ungetch() will convert *c* into an unsigned char and push that character back onto the input stream. Only one character of push-back is guaranteed to work, more may be possible depending on system resources.

RETURN VALUES

The functions **getch**(), **wgetch**(), **mvgetch**(), and **mvwgetch**() will return the value of the key pressed or ERR in the case of an error or a timeout. Additionally, if **keypad**(*TRUE*) has been called on a window, then it may return one of the following values:

Termcap entry	getch Return Value	Key Function
!1	KEY_SSAVE	Shift Save
!2	KEY_SSUSPEND	Shift Suspend
!3	KEY_SUNDO	Shift Undo
#1	KEY_SHELP	Shift Help
#2	KEY_SHOME	Shift Home
#3	KEY_SIC	Shift Insert Character
#4	KEY_SLEFT	Shift Left Arrow
%0	KEY_REDO	Redo
%1	KEY_HELP	Help
%2	KEY_MARK	Mark
%3	KEY_MESSAGE	Message
%4	KEY_MOVE	Move
%5	KEY_NEXT	Next Object
%6	KEY_OPEN	Open
%7	KEY_OPTIONS	Options
%8	KEY_PREVIOUS	Previous Object

%9	KEY_PRINT	Print
%a	KEY_SMESSAGE	Shift Message
%b	KEY_SMOVE	Shift Move
%c	KEY_SNEXT	Shift Next Object
%d	KEY_SOPTIONS	Shift Options
%e	KEY_SPREVIOUS	Shift Previous Object
%f	KEY_SPRINT	Shift Print
%g	KEY_SREDO	Shift Redo
%h	KEY_SREPLACE	Shift Replace
%i	KEY_SRIGHT	Shift Right Arrow
%j	KEY_SRSUME	Shift Resume
&0	KEY_SCANCEL	Shift Cancel
&1	KEY_REFERENCE	Reference
&2	KEY_REFRESH	Refresh
&3	KEY_REPLACE	Replace
&4	KEY_RESTART	Restart
&5	KEY_RESUME	Resume
&6	KEY_SAVE	Save
&7	KEY_SUSPEND	Suspend
&8	KEY_UNDO	Undo
&9	KEY_SBEG	Shift Begin
∗0	KEY_SFIND	Shift Find
∗1	KEY_SCOMMAND	Shift Command
∗2	KEY_SCOPY	Shift Copy
∗3	KEY_SCREATE	Shift Create
∗4	KEY_SDC	Shift Delete Character
∗5	KEY_SDL	Shift Delete Line
∗6	KEY_SELECT	Select
∗7	KEY_SEND	Shift End
∗8	KEY_SEOL	Shift Clear to EOL
∗9	KEY_SEXIT	Shift Exit
@0	KEY_FIND	Find
@1	KEY_BEG	Begin
@2	KEY_CANCEL	Cancel
@3	KEY_CLOSE	Close
@4	KEY_COMMAND	Command
@5	KEY_COPY	Copy
@6	KEY_CREATE	Create
@7	KEY_END	End
@8	KEY_ENTER	Enter
@9	KEY_EXIT	Exit
F1	KEY_F(11)	Function Key 11
F2	KEY_F(12)	Function Key 12
F3	KEY_F(13)	Function Key 13
F4	KEY_F(14)	Function Key 14
F5	KEY_F(15)	Function Key 15
F6	KEY_F(16)	Function Key 16
F7	KEY_F(17)	Function Key 17
F8	KEY_F(18)	Function Key 18

F9	KEY_F(19)	Function Key 19
FA	KEY_F(20)	Function Key 20
FB	KEY_F(21)	Function Key 21
FC	KEY_F(22)	Function Key 22
FD	KEY_F(23)	Function Key 23
FE	KEY_F(24)	Function Key 24
FF	KEY_F(25)	Function Key 25
FG	KEY_F(26)	Function Key 26
FH	KEY_F(27)	Function Key 27
FI	KEY_F(28)	Function Key 28
FJ	KEY_F(29)	Function Key 29
FK	KEY_F(30)	Function Key 30
FL	KEY_F(31)	Function Key 31
FM	KEY_F(32)	Function Key 32
FN	KEY_F(33)	Function Key 33
FO	KEY_F(34)	Function Key 34
FP	KEY_F(35)	Function Key 35
FQ	KEY_F(36)	Function Key 36
FR	KEY_F(37)	Function Key 37
FS	KEY_F(38)	Function Key 38
FT	KEY_F(39)	Function Key 39
FU	KEY_F(40)	Function Key 40
FV	KEY_F(41)	Function Key 41
FW	KEY_F(42)	Function Key 42
FX	KEY_F(43)	Function Key 43
FY	KEY_F(44)	Function Key 44
FZ	KEY_F(45)	Function Key 45
Fa	KEY_F(46)	Function Key 46
Fb	KEY_F(47)	Function Key 47
Fc	KEY_F(48)	Function Key 48
Fd	KEY_F(49)	Function Key 49
Fe	KEY_F(50)	Function Key 50
Ff	KEY_F(51)	Function Key 51
Fg	KEY_F(52)	Function Key 52
Fh	KEY_F(53)	Function Key 53
Fi	KEY_F(54)	Function Key 54
Fj	KEY_F(55)	Function Key 55
Fk	KEY_F(56)	Function Key 56
Fl	KEY_F(57)	Function Key 57
Fm	KEY_F(58)	Function Key 58
Fn	KEY_F(59)	Function Key 59
Fo	KEY_F(60)	Function Key 60
Fp	KEY_F(61)	Function Key 61
Fq	KEY_F(62)	Function Key 62
Fr	KEY_F(63)	Function Key 63
K1	KEY_A1	Upper left key in keypad
K2	KEY_B2	Centre key in keypad
K3	KEY_A3	Upper right key in keypad
K4	KEY_C1	Lower left key in keypad

K5	KEY_C3	Lower right key in keypad
Km	KEY_MOUSE	Mouse Event
k0	KEY_F0	Function Key 0
k1	KEY_F(1)	Function Key 1
k2	KEY_F(2)	Function Key 2
k3	KEY_F(3)	Function Key 3
k4	KEY_F(4)	Function Key 4
k5	KEY_F(5)	Function Key 5
k6	KEY_F(6)	Function Key 6
k7	KEY_F(7)	Function Key 7
k8	KEY_F(8)	Function Key 8
k9	KEY_F(9)	Function Key 9
k;	KEY_F(10)	Function Key 10
kA	KEY_IL	Insert Line
ka	KEY_CATAB	Clear All Tabs
kB	KEY_BTAB	Back Tab
kb	KEY_BACKSPACE	Backspace
kC	KEY_CLEAR	Clear
kD	KEY_DC	Delete Character
kd	KEY_DOWN	Down Arrow
kE	KEY_EOL	Clear to End Of Line
kF	KEY_SF	Scroll Forward one line
kH	KEY_LL	Home Down
kh	KEY_HOME	Home
kI	KEY_IC	Insert Character
kL	KEY_DL	Delete Line
kl	KEY_LEFT	Left Arrow
kM	KEY_EIC	Exit Insert Character Mode
kN	KEY_NPAGE	Next Page
kP	KEY_PPAGE	Previous Page
kR	KEY_SR	Scroll One Line Back
kr	KEY_RIGHT	Right Arrow
kS	KEY_EOS	Clear to End Of Screen
kT	KEY_STAB	Set Tab
kt	KEY_CTAB	Clear Tab
ku	KEY_UP	Up Arrow

Note that not all terminals are capable of generating all the keycodes listed above nor are termcap entries normally configured with all the above capabilities defined.

Other functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

Functions returning pointers will return `NULL` if an error is detected.

SEE ALSO

`curses_cursor`(3), `curses_keyname`(3), `curses_refresh`(3), `curses_tty`(3), `termcap`(5)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

NOTES

The **keyok**() and **define_key**() functions are implementations of extensions made by the NCurses library to the Curses standard. Portable implementations should avoid the use of these functions.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_insdelln, **insdelln**, **winsdelln** — curses insert or delete lines routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
insdelln(int n);

int
winsdelln(WINDOW *win, int n);
```

DESCRIPTION

These functions insert or delete lines on `stdscr` or on the specified window.

If **insdelln**() is called with a positive number in *n*, then the specified number of lines are inserted before the current line on `stdscr`. The last *n* lines of the screen are no longer displayed. If *n* is negative, then *n* lines are deleted from `stdscr`, starting at the current line. The last *n* lines of `stdscr` are cleared.

The **winsdelln**() function is the same as the **insdelln**() function, excepting that lines are inserted or deleted from the window specified by *win*.

If a scrolling region has been set with the **setscrreg**() or **wsetscrreg**() functions and the current cursor position is inside the scrolling region, then only the lines from the current line to the bottom of the scrolling region are affected.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_deleteln`(3), `curses_insertln`(3), `curses_scroll`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_insert, **insch**, **winsch**, **mvinsch**, **mvwinsch** — curses insert characters routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
insch(chtype ch);

int
winsch(WINDOW *win, chtype ch);

int
mvinsch(int y, int x, chtype ch);

int
mvwinsch(WINDOW *win, int y, int x, chtype ch);
```

DESCRIPTION

These functions insert characters on `stdscr` or on the specified window.

The **insch**() function inserts the character given in *ch* at the current cursor position on `stdscr`. The cursor is not advanced and wrapping is not performed.

The **winsch**() function is the same as the **insch**() function, excepting that the character is inserted on the window specified by *win*.

The **mvinsch**() and **mvwinsch**() functions are the same as the **insch**() and **insch**() functions, respectively, excepting that **wmove**() is called to move the cursor to the position specified by *y*, *x* before the character is inserted on the window.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_addch`(3), `curses_cursor`(3), `curses_delch`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_insertln, **insertln**, **winsertln** — curses insert single line routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
insertln(void);

int
winsertln(WINDOW *win);
```

DESCRIPTION

These functions insert a single line on `stdscr` or on the specified window.

The **insertln**() function inserts a blank line before the current line on `stdscr`. The current line and all lines below are moved down one line away from the cursor and the bottom line of the window is lost.

The **winsertln**() function is the same as the **insertln**() function, excepting that the line is inserted on the window *win*.

If a scrolling region has been set with the **setscrreg**() or **wsetscrreg**() functions and the current cursor position is inside the scrolling region, then only the lines from the current line to the bottom of the scrolling region are moved down and the bottom line of the scrolling region lost.

The functions **insertln**() and **winsertln**(*win*) are equivalent to **insdelln**(*1*) and **winsdelln**(*win*, *1*), respectively.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_deleteln`(3), `curses_insdelln`(3), `curses_scroll`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_keyname, **keyname** — curses report key name routine

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

char *
keyname(int key);
```

DESCRIPTION

The function **keyname**() generates a character string containing a description of the key specified in *key*.

The string is formatted according to the following table:

Description	Key range	String format	
Control character	0 - 31	^X	
Visible character	32 - 126	X	
Delete character	127	^?	
Meta + control character		128 - 158	M-^X
Meta + visible character		159 - 254	M-X
Meta + delete character		255	M-^?
Named key	KEY_MIN - KEY_MAX	KEY_EXIT	
Unknown key		-1	

SEE ALSO

curses_input(3)

NOTE

The return value of **keyname**() is a static buffer, which will be overwritten on a subsequent call.

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

These functions first appeared in NetBSD 2.0.

NAME

curses_line, **hline**, **whline**, **vline**, **wvline**, **mvhline**, **mvwhline**, **mvvline**, **mvwvline** — curses draw lines on windows routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
hline(chtype ch, int n);

int
whline(WINDOW *win, chtype ch, int n);

int
mvhline(int y, int x, chtype ch, int n);

int
mvwvline(WINDOW *win, int y, int x, chtype c, int n);

int
vline(chtype c, int n);

int
wvline(WINDOW *win, chtype c, int n);

int
mvvline(int y, int x, chtype ch, int n);

int
mvwhline(WINDOW *win, int y, int x, chtype c, int n);
```

DESCRIPTION

These functions draw lines on `stdscr` or on the specified window.

The **hline**() function draws a horizontal line of the character *ch* on `stdscr` starting at the current cursor position and extending for *n* characters, or until the right hand side of `stdscr` is reached. If the text portion of *ch* is 0 then the line will be drawn with the `ACS_HLINE` character.

The **whline**() function is the same as the **hline**() function, excepting that the line is drawn in the window specified by *win*.

The **vline**() function draws a vertical line of character *ch* on `stdscr` starting at the current cursor position and moving down until either *n* characters have been drawn or the bottom of `stdscr` is reached. If the text portion of *ch* is 0 then the line will be drawn with the `ACS_VLINE` character.

The **wvline**() function is the same as the **vline**() function, excepting that the line is drawn on the given window.

The **mvhline**(), **mvwhline**(), **mvvline**() and **mvwvline**() functions are the same as the **hline**(), **whline**(), **vline**() and **wvline**() functions, respectively, excepting that **wmove**() is called to move the cursor to the position specified by *y*, *x* before the line is drawn on the window.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_border`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_pad, **newpad**, **subpad**, **prefresh**, **pnoutrefresh** — curses pad routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

WINDOW *
newpad(int lines, int cols);

WINDOW *
subpad(WINDOW *pad, int lines, int cols, int begin_y, int begin_x);

int
prefresh(WINDOW *pad, int pbeg_y, int pbeg_x, int sbeg_y, int sbeg_x,
    int smax_y, int smax_x);

int
pnoutrefresh(WINDOW *pad, int pbeg_y, int pbeg_x, int sbeg_y, int sbeg_x,
    int smax_y, int smax_x);
```

DESCRIPTION

These functions create and display pads on the current screen.

The **newpad**() function creates a new pad of size *lines*, *cols*.

subpad() is similar to **newpad**() excepting that the size of the subpad is bounded by the parent pad *pad*. The subpad shares internal data structures with the parent pad and will be refreshed when the parent pad is refreshed. The starting column and row *begin_y*, *begin_x* are relative to the parent pad origin.

The **pnoutrefresh**() function performs the internal processing required by curses to determine what changes need to be made to synchronise the internal screen buffer and the terminal but does not modify the terminal display. A rectangular area of the pad starting at column and row *pbeg_y*, *pbeg_x* is copied to the corresponding rectangular area of the screen buffer starting at column and row *sbeg_y*, *sbeg_x* and extending to *smax_y*, *smax_x*.

The **prefresh**() function causes curses to propagate changes made to the pad specified by *pad* to the terminal display. A rectangular area of the pad starting at column and row *pbeg_y*, *pbeg_x* is copied to the corresponding rectangular area of the terminal starting at column and row *sbeg_y*, *sbeg_x* and extending to *smax_y*, *smax_x*.

The **pnoutrefresh**() and **doupdate**() functions can be used together to speed up terminal redraws by deferring the actual terminal updates until after a batch of updates to multiple pads has been done.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_refresh`(3), `curses_window`(3)

NOTES

The **subpad**() function is similar to the derwin(3) function, and not the subwin(3) function.

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_print, **printw**, **wprintw**, **mvprintw**, **mvwprintw**, **unctrl** — curses print formatted strings on windows routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
printw(const char *fmt, ...);

int
wprintw(WINDOW *win, const char *fmt, ...);

int
mvprintw(int y, int x, const char *fmt, ...);

int
mvwprintw(WINDOW *win, int y, int x, const char *fmt, ...);

char *
unctrl(chtype ch);
```

DESCRIPTION

These functions print formatted strings on `stdscr` or on the specified window.

The **printw**() function formats and prints its arguments on `stdscr`. The behaviour is the same as that of **printf**().

The **wprintw**() function is the same as the **printw**() function, excepting that the resulting output is printed on the window specified by *win*.

The **mvprintw**() and **mvwprintw**() functions are the same as the **printw**() and **wprintw**() functions, respectively, excepting that **wmove**() is called to move the cursor to the position specified by *y*, *x* before the output is printed on the window.

The **unctrl**() function returns a printable string representation of the character *ch*. If *ch* is a control character then it will be converted to the form ^Y.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_cursor`(3), `curses_scanw`(3), `printf`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_refresh, **refresh**, **wrefresh**, **wnoutrefresh**, **doupdate**, **leaveok**, **flushok** — curses terminal update routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
refresh(void);

int
wrefresh(WINDOW *win);

int
wnoutrefresh(WINDOW *win);

int
doupdate(void);

int
leaveok(WINDOW *win, boolf flag);

int
flushok(WINDOW *win, boolf flag);
```

DESCRIPTION

These functions update the terminal with the contents of `stdscr` or of the specified window(s).

The **refresh**() function causes curses to propagate changes made to `stdscr` to the terminal display. Any changes made to subwindows of `stdscr` are also propagated.

The **wrefresh**() function is the same as the **refresh**() function, excepting that changes are propagated to the terminal from the window specified by *win*.

The **wnoutrefresh**() function performs the internal processing required by curses to determine what changes need to be made to synchronise the internal screen buffer and the terminal but does not modify the terminal display.

The **doupdate**() function updates the terminal display to match the internal curses representation of the display.

The **wnoutrefresh**() and **doupdate**() functions can be used together to speed up terminal redraws by deferring the actual terminal updates until after a batch of updates to multiple windows has been done.

The **refresh**() function is equivalent to **wnoutrefresh**(*stdscr*) followed by **doupdate**().

The **leaveok**() function determines whether refresh operations may leave the screen cursor in an arbitrary position on the screen. Setting *flag* to FALSE ensures that the screen cursor is positioned at the current cursor position after a refresh operation has taken place.

The **flushok**() function is used to determine whether or not the screen's output file descriptor will be flushed on refresh. Setting *flag* to TRUE will cause the output to be flushed.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_pad`(3), `curses_touch`(3), `getch`(3)

NOTES

Calling **`wrefresh`**() on a new, unchanged window has no effect.

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_scanw, **scanw**, **wscanw**, **mvscanw**, **mvwscanw** — curses read formatted data from screen routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
scanw(const char *fmt, ...);

int
wscanw(WINDOW *win, const char *fmt, ...);

int
mvscanw(int y, int x, const char *fmt, ...);

int
mvwscanw(WINDOW *win, int y, int x, const char *fmt, ...);
```

DESCRIPTION

These functions read formatted data from `stdscr` or from the specified window.

The **scanw**() function is the same as the **scanf**() function, excepting that the input data stream is read from the current cursor position on `stdscr`,

The **wscanw**() function is the same as the **scanw**() function, excepting that the data stream is read from the window specified by *win*.

The **mvscanw**() and **mvwscanw**() functions are the same as the **scanw**() and **mvscanw**() functions, respectively, excepting that **wmove**() is called to move the cursor to the position specified by *y*, *x* before the data is read from the window.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_cursor`(3), `curses_print`(3), `scanf`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_screen, **newterm**, **set_term**, **delscreen**, **endwin**, **initscr**, **isendwin**, **resizeterm**, **setterm** — curses terminal and screen routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

SCREEN *
newterm(char *type, FILE *outfd, FILE *infd);

SCREEN *
set_term(SCREEN *screen);

void
delscreen(SCREEN *screen);

int
endwin(void);

WINDOW *
initscr(void);

bool
isendwin(void);

int
resizeterm(int lines, int cols);

int
setterm(char *name);

extern int LINES;

extern int COLS;
```

DESCRIPTION

These functions initialize terminals and screens.

The **newterm**() function initialises the curses data structures and pointers ready for use by curses. The *type* argument points to a termcap(5) capability name, or it may be NULL in which case the TERM environment variable is used. The *outfd* and *infd* are the output and input file descriptors for the terminal. The **newterm**() function must only be called once per terminal.

The **set_term**() function can be used to switch between the screens defined by calling **newterm**(), a pointer to the previous screen structure that was in use will be returned on success.

Calling **delscreen**() will destroy the given screen and free all allocated resources.

Calling **endwin**() will end the curses session and restore the saved terminal settings.

The curses session must be initialised by calling **initscr**() which saves the current terminal state and sets up the terminal and internal data structures to support the curses application. This function call must be, with few exceptions, the first Curses library call made. The exception to this rule is the **newterm**() call which may be called prior to **initscr**(). The size of the curses screen is determined by checking the tty(4) size and then the termcap(5) entries for the terminal type. If the environment variables *LINES* or *COLS* are set, then these will be used instead.

When either **newterm**() or **initscr**() are called, the Curses library sets up signal handlers for SIGTSTP and SIGWINCH. If a signal handler is already installed for SIGWINCH, this will also be called when the Curses library handler is called.

The **isendwin**() function can be used to determine whether or not a refresh of the screen has occurred since the last call to **endwin**().

The size of the screen may be changed by calling **resizeterm**() with the updated number of lines and columns. This will resize the curses internal data structures to accommodate the changed terminal geometry. The curscr and stdscr windows and any of their subwindows will be resized to fit the new screen size. The application must redraw the screen after a call to **resizeterm**().

The **setterm**() function sets the terminal type for the current screen to the one passed, initialising all the curses internal data structures with information related to the named terminal. The *name* argument must be a valid name or alias in the termcap(5) database for this function to succeed.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following values:

OK The function completed successfully.
ERR An error occurred in the function.

SEE ALSO

curses_window(3), tty(4), termcap(5), signal(7)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD. The **resizeterm**() function is a *ncurses* extension to the Curses library and was added in NetBSD 1.6.

NAME

curses_scroll, **scrl**, **wscrl** **scroll**, **scrollok**, **setscrreg**, **wsetscrreg** — curses window scrolling routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
scrl(int n);

int
wscrl(WINDOW *win, int n);

int
scroll(WINDOW *win);

int
scrollok(WINDOW *win, boolf flag);

int
setscrreg(int top, int bottom);

int
wsetscrreg(WINDOW *win, int top, int bottom);
```

DESCRIPTION

These functions scroll areas on `stdscr` or on the specified window.

The **scrl**() function scrolls `stdscr` by *n* lines. If *n* is positive then then `stdscr` is scrolled up. *n* lines are lost from the top of `stdscr` and *n* blank lines are inserted at the bottom. If *n* is negative then `stdscr` is scrolled down. *n* blank lines are inserted at the top of `stdscr` and *n* lines are lost from the bottom.

The **wscrl**() function is the same as the **scrl**() function, excepting that it scrolls the window specified by *win*.

The **scroll**() function scrolls the window *win* up by one line.

The scrolling behaviour of a window can be controlled by using the **scrollok**() function. If the *flag* argument is TRUE then a line wrap at the bottom of the window will cause the window to be scrolled up one line, if *flag* is FALSE then lines that would force a scroll will be truncated.

The **setscrreg**() function sets up a software scrolling region on `stdscr` which will define a region of the screen that will be scrolled. The scrolling of this region is also controlled by the **scrollok**() function.

The **wsetscrreg**() function does the same as the **setscrreg**() function, except that the scrolling region is set on the window specified by *win*.

If a scrolling region has been set with the **setscrreg**() or **wsetscrreg**() functions and the current cursor position is inside the scrolling region, then only the area inside the scrolling region is scrolled.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following values:

`OK` The function completed successfully.
`ERR` An error occurred in the function.

SEE ALSO

`curses_deleteln`(3), `curses_insdelln`(3), `curses_insertln`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_standout, **standout**, **standend**, **wstandout**, **wstandend** — curses standout attribute manipulation routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
standout(void);

int
standend(void);

int
wstandout(void);

int
wstandend(void);
```

DESCRIPTION

These functions manipulate the standout attribute on `stdscr` or on the specified window.

The **standout**() function turns on the standout attribute on `stdscr`. The **standend**() function turns off the standout attribute on `stdscr`.

The **wstandout**() and **wstandend**() functions are equivalent to **standout**() and **standend**(), respectively, excepting that the attribute is manipulated on the window specified by *win*.

The **standout**() and **standend**() functions are equivalent to **attron**(*A_STANDOUT*) and **attroff**(*A_STANDOUT*), respectively.

RETURN VALUES

These functions always return 1.

SEE ALSO

`curses_attributes`(3), `curses_underscore`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

BUGS

On modern terminals that support other attributes, there is no difference between characters displayed with the standout attribute set and those displayed with one of the other attributes set (usually bold). It is best to avoid using standout if the terminal supports other attributes.

NAME

curses_termcap, **fullname**, **getcap**, **longname** — curses termcap querying routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

char *
fullname(char *termbuf, char *name);

char *
getcap(char *name);

char *
longname(void);
```

DESCRIPTION

The **fullname**() function takes the termcap entry in *termbuf* and copies the full name of the terminal from the entry into the target variable *name*. The full name of the terminal is assumed to be the last alias in the termcap entry name. It is assumed that the *name* variable has sufficient storage to hold the full name of the terminal.

A termcap entry can be retrieved by calling the **getcap**() function with the name of the capability in *name*. The matching capability string for the terminal is returned.

The **longname**() function returns a verbose description of the terminal which is taken from the last name alias in the termcap description for the terminal. This string will be at most 128 characters long and will only be defined after a call to **initscr**() or **newterm**().

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following values:

OK The function completed successfully.
ERR An error occurred in the function.

SEE ALSO

termcap(5)

STANDARDS

The Curses Library (libcurses, −lcurses) library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_touch, **touchline**, **touchoverlap**, **touchwin**, **untouchwin**, **wtouchln**, **is_linetouched**, **is_wintouched**, **redrawwin**, **wredrawln** — curses window modification routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
touchline(WINDOW *win, int start, int count);

int
touchoverlap(WINDOW *win1, WINDOW *win2);

int
touchwin(WINDOW *win);

int
untouchwin(WINDOW *win);

int
wtouchln(WINDOW *win, int line, int n, boolf changed);

bool
is_linetouched(WINDOW *win, int line);

bool
is_wintouched(WINDOW *win);

int
redrawwin(WINDOW *win);

int
wredrawln(WINDOW *win, int line, int n);
```

DESCRIPTION

These functions mark lines and windows as modified and check the modification status of lines and windows.

The **touchline**() function marks *count* lines starting from *start* in window *win* as having been modified. These characters will be synced to the terminal on the next call to **wrefresh**().

The **touchoverlap**() function marks the portion of *win2* that overlaps *win1* as being modified.

The **touchwin**() function marks the entire window *win* as having been modified. Conversely, the **untouchwin**() function marks the window *win* as being unmodified, so that any changes made to that window will not be synced to the terminal during a **wrefresh**().

The **wtouchln**() function performs one of two operations on *n* lines starting at *line* in the given window. If *changed* is 1 then the given line range is marked as being modified, if *changed* is 0 then the given line range is set to being unmodified.

The **is_linetouched**() function returns TRUE if *line* in window *win* has been modified since the last refresh was done, otherwise FALSE is returned.

is_wintouched() returns TRUE if the window *win* has been modified since the last refresh, otherwise FALSE is returned.

The **redrawwin**() function marks the entire window *win* as having been corrupted. Is is equivalent to the **touchwin**() function.

The **wredrawln**() function marks *n* lines starting at *line* in the given window as corrupted. It is equivalent to **wtouchln**(*win*, *line*, *n*, *1*).

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following values:

OK The function completed successfully.
ERR An error occurred in the function.

SEE ALSO

curses_refresh(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_tty, **beep**, **flash**, **curs_set**, **def_prog_mode**, **reset_prog_mode**, **def_shell_mode**, **reset_shell_mode**, **echo**, **noecho**, **delay_output**, **erasechar**, **flushinp**, **gettmode**, **halfdelay**, **has_ic**, **has_il**, **idcok**, **idlok**, **intrflush**, **noqiflush**, **qiflush**, **killchar**, **meta**, **napms**, **nl**, **nonl**, **cbreak**, **nocbreak**, **raw**, **noraw**, **savetty**, **resetty** — curses terminal manipulation routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
beep(void);

int
flash(void);

int
curs_set(int visibility);

int
def_prog_mode(void);

int
reset_prog_mode(void);

int
def_shell_mode(void);

int
reset_shell_mode(void);

int
echo(void);

int
noecho(void);

int
delay_output(int ms);

char
erasechar(void);

int
flushinp(void);

int
gettmode(void);

int
has_ic(void);

int
has_il(void);
```

int
idcok(*WINDOW *win*, *boolf flag*);

int
idlok(*WINDOW *win*, *boolf flag*);

int
intrflush(*WINDOW *win*, *boolf flag*);

void
noqiflush(*void*);

void
qiflush(*void*);

char
killchar(*void*);

int
meta(*WINDOW *win*, *boolf flag*);

int
napms(*int ms*);

int
nl(*void*);

int
nonl(*void*);

int
cbreak(*void*);

int
nocbreak(*void*);

int
halfdelay(*int*);

int
raw(*void*);

int
noraw(*void*);

int
savetty(*void*);

int
resetty(*void*);

DESCRIPTION

These functions manipulate curses terminal settings.

The **beep**() function rings the terminal bell, if this is possible. Failing that, the terminal screen will be flashed. If neither of these are possible, then no action will be taken. **flash**() will flash the terminal screen if possible. Failing that, the terminal bell will be rung. If neither of these are possible then no action will be taken.

The cursor visibility can be set by calling **curs_set**(). The following visibility settings are valid for **curs_set**():

Visibility	Effect
0	cursor is invisible.
1	cursor is normal visibility
2	cursor is high visibility

A successful call to **curs_set**() will return the previous visibility setting for the cursor.

The **delay_output**() function pauses the output to the terminal by sending the appropriate number of terminal pad characters such that the transmission time of the pad characters will take *ms* milliseconds.

Calling **def_prog_mode**() will cause the current terminal curses setting to be saved. A subsequent call to **reset_prog_mode**(), will restore the saved settings. This is useful when calls to external programs are made that may reset the terminal characteristics.

The **def_shell_mode**() function saves the current terminal line settings. These settings are the ones that will be restored when the curses application exits. Conversely, **reset_shell_mode**() will save the current terminal curses settings for later restoration and restores the previously saved terminal line settings.

The **echo**() function turns on curses echo mode, characters entered will be echoed to the terminal by curses. The **noecho**() function disables this feature.

The current erase character for the terminal can be determined by calling the **erasechar**() function.

The **flushinp**() function discards any pending input for the current screen.

The modes for the current terminal can be reset by calling **gettmode**(), this will perform the initialisation on the terminal that is normally done by curses at start up.

The **has_ic**() function returns either TRUE or FALSE depending on whether or not the terminal has a insert character capability or not. Similarly the **has_il**() function does the same test but for a insert line capability.

The use of the insert character capability in curses operations can be enabled or disabled by calling **idcok**() on the desired window. Similarly, the use of the insert line capability can be controlled using the **idlok**() function.

The **intrflush**() function controls whether or not a flush of the input buffer is performed when an interrupt key (kill, suspend or quit) is pressed. The *win* parameter is ignored. The **noqiflush**() function is equivalent to **intrflush**(*stdscr*, *FALSE*). The **qiflush**() function is equivalent to **intrflush**(*stdscr*, *TRUE*).

The character that performs the line kill function can be determined by calling the **killchar**() function.

The **meta**() function turns on and off the generation of 8 bit characters by the terminal, if *flag* is FALSE then only 7 bit characters will be returned, if *flag* is TRUE then 8 bit characters will be returned by the terminal.

The **napms**() causes the application to sleep for the number of milliseconds specified by *ms*.

Calling **nl**() will cause curses to map all carriage returns to newlines on input, this functionality is enabled by default. The **nonl**() function disables this behaviour.

The **cbreak**() function will put the terminal into cbreak mode, which means that characters will be returned one at a time instead of waiting for a newline character, line discipline processing will be performed. The **nocbreak**() function disables this mode.

Calling **halfdelay**() puts the terminal into the same mode as **cbreak**() with the exception that if no character is received within the specified number of tenths of a second then the input routine will return ERR. This mode can be cancelled by calling **nocbreak**(). The valid range for the timeout is from 1 to 255 tenths of a second.

The **noraw**() function sets the input mode for the current terminal into Cooked mode, that is input character translation and signal character processing is performed. The **raw**() function puts the terminal into Raw mode, no input character translation is done nor is signal character processing.

The terminal tty flags can be saved by calling **savetty**() and may be restored by calling **resetty**(), the use of these functions is discouraged as they may cause the terminal to be put into a state that is incompatible with curses operation.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following values:

OK The function completed successfully.
ERR An error occurred in the function.

SEE ALSO

getch(3), termios(4)

NOTES

The **idcok**() and **idlok**() currently have no effect on the curses code at all, currently curses will always use the terminal insert character and insert line capabilities if available.

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

curses_underscore, **underscore**, **underend**, **wunderscore**, **wunderend** — curses underscore attribute manipulation routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
underscore(void);

int
underend(void);

int
wunderscore(void);

int
wunderend(void);
```

DESCRIPTION

These functions manipulate the underscore attribute on `stdscr` or on the specified window.

The **underscore**() function turns on the underscore attribute on `stdscr`. The **underend**() function turns off the underscore attribute on `stdscr`.

The **wunderscore**() and **wunderend**() functions are equivalent to **underscore**() and **underend**(), respectively, excepting that the attribute is manipulated on the window specified by *win*.

The **underscore**() and **underend**() functions are equivalent to **wattron**(*A_UNDERLINE*) and **wattroff**(*A_UNDERLINE*), respectively.

RETURN VALUES

These functions always return 1.

SEE ALSO

`curses_attributes`(3), `curses_standout`(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

These functions first appeared in NetBSD 1.5.

NAME

curses_window, **copywin**, **dupwin**, **delwin**, **derwin**, **mvwin**, **mvderwin**, **newwin**, **overlay**, **overwrite**, **subwin**, **wresize** — curses window routines

LIBRARY

Curses Library (libcurses, −lcurses)

SYNOPSIS

```
#include <curses.h>

int
copywin(WINDOW *source, WINDOW *dest, int sminrow, int smincol, int dminrow,
    int dmincol, int dmaxrow, int dmaxcol, int overlay);

WINDOW *
dupwin(WINDOW *win);

WINDOW *
derwin(WINDOW *win, int lines, int cols, int y, int x);

int
delwin(WINDOW *win);

int
mvwin(WINDOW *win, int y, int x);

int
mvderwin(WINDOW *win, int y, int x);

WINDOW *
newwin(int lines, int cols, int begin_y, int begin_x);

WINDOW *
subwin(WINDOW *win, int lines, int cols, int begin_y, int begin_x);

int
overlay(WINDOW *source, WINDOW *dest);

int
overwrite(WINDOW *source, WINDOW *dest);

int
wresize(WINDOW *win, int lines, int cols);
```

DESCRIPTION

These functions create, modify and delete windows on the current screen.

The contents of a window may be copied to another window by using the **copywin**() function, a section of the destination window *dest* bounded by *(dminrow*, *dmincol*) and *(dmaxrow*, *dmaxcol*) will be overwritten with the contents of the window *source* starting at the coordinates *(sminrow*, *smincol*). If the *overlay* flag is TRUE then only non-blank characters from *source* will be copied to *dest*, if *overlay* is FALSE then all characters from *source* will be copied to *dest*. If the bounding rectangles of either the source or the destination windows lay outside the maximum size of the respective windows then the size of the window copied will be adjusted to be within the bounds of both the source and destination windows.

The **dupwin**() function creates an exact duplicate of *win* and returns a pointer to it.

Calling **derwin**() will create a subwindow of *win* in the same manner as **subwin**() excepting that the starting column and row *y*, *x* are relative to the parent window origin.

A window may deleted and all resources freed by calling the **delwin**() function with the pointer to the window to be deleted in *win*.

A window can be moved to a new position by calling the **mvwin**() function. The *y* and *x* positions are the new origin of the window on the screen. If the new position would cause the any part of the window to lie outside the screen, it is an error and the window is not moved.

A subwindow can be moved relative to the parent window by calling the **mvderwin**() function, the *y* and *x* positions are relative to the origin of the parent window. If the given window in *win* is not a subwindow then an error will be returned. If the new position would cause the any part of the window to lie outside the parent window, it is an error and the window is not moved.

The **newwin**() function creates a new window of size *lines*, *cols* with an origin at *begin_y*, *begin_x*. If *lines* is less than or equal to zero then the number of rows for the window is set to LINES - *begin_x* + *lines*. Similarly if *cols* is less than or equal to zero then the number of columns for the window is set to COLS - *begin_y* + *cols*.

subwin() is similar to **newwin**() excepting that the size of the subwindow is bounded by the parent window *win*. The subwindow shares internal data structures with the parent window and will be refreshed when the parent window is refreshed. The subwindow inherits the background character and attributes of the parent window.

The **overlay**() function copies the contents of the source window *source* to the destination window *dest*, only the characters that are not the background character in the source window are copied to the destination. The windows need not be the same size, only the overlapping portion of both windows will be copied. The **overwrite**() function performs the same functions as **overlay**() excepting that characters from the source window are copied to the destination without exception.

wresize() resizes the specified window to the new number of lines and columns given, all internal curses structures are resized. Any subwindows of the specified window will also be resized if any part of them falls outside the new parent window size. The application must redraw the window after it has been resized. Note that curscr and stdscr can not be resized to be larger than the size of the screen.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following values:

OK The function completed successfully.
ERR An error occurred in the function.

SEE ALSO

curses_pad(3), curses_fileio(3), curses_screen(3)

STANDARDS

The NetBSD Curses library complies with the X/Open Curses specification, part of the Single Unix Specification.

HISTORY

The Curses package appeared in 4.0BSD.

NAME

cuserid — get user name

LIBRARY

Compatibility Library (libcompat, −lcompat)

SYNOPSIS

```
#include <stdio.h>

char *
cuserid(char *buf);
```

DESCRIPTION

This interface is available from the compatibility library, libcompat and has been obsoleted by `getlogin`**(2).**

The **cuserid**() function returns a character string representation of the user name associated with the effective user ID of the calling process.

If *buf* is not the `NULL` pointer, the user name is copied into the memory referenced by *buf*. The argument *buf* is assumed to point to an array at least `L_cuserid` (as defined in the include file ⟨`stdio.h`⟩) bytes long. Otherwise, the user name is copied to a static buffer.

RETURN VALUES

If *buf* is not the `NULL` pointer, *buf* is returned; otherwise the address of the static buffer is returned.

If the user name could not be determined, if *buf* is not the `NULL` pointer, the null character '\0' will be stored at **buf*; otherwise the `NULL` pointer is returned.

SEE ALSO

`getlogin`(2), `getpwent`(3)

STANDARDS

The **cuserid**() function conforms to IEEE Std 1003.1-1988 ("POSIX.1").

BUGS

Due to irreconcilable differences in historic implementations, **cuserid**() was removed from the ISO/IEC 9945-1:1990 ("POSIX.1") standard. This implementation exists purely for compatibility with existing programs. New programs should use one of the following three alternatives to obtain the user name:

1. **getlogin**() to return the user's login name.
2. **getpwuid**(**geteuid**()) to return the user name associated with the calling process' effective user ID.
3. **getpwuid**(**getuid**()) to return the user name associated with the calling process' real user ID.

NAME

d2i_ASN1_OBJECT, i2d_ASN1_OBJECT − ASN1 OBJECT IDENTIFIER functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/objects.h>

ASN1_OBJECT *d2i_ASN1_OBJECT(ASN1_OBJECT **a, unsigned char **pp, long length);
int i2d_ASN1_OBJECT(ASN1_OBJECT *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an ASN1 OBJECT IDENTIFIER.

Othewise these behave in a similar way to *d2i_X509()* and *i2d_X509()* described in the *d2i_X509* (3) manual page.

SEE ALSO

d2i_X509 (3)

HISTORY

TBA

NAME

d2i_DHparams, i2d_DHparams − PKCS#3 DH parameter functions.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dh.h>

DH *d2i_DHparams(DH **a, unsigned char **pp, long length);
int i2d_DHparams(DH *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode PKCS#3 DH parameters using the DHparameter structure described in PKCS#3.

Othewise these behave in a similar way to *d2i_X509()* and *i2d_X509()* described in the *d2i_X509* (3) manual page.

SEE ALSO

d2i_X509 (3)

HISTORY

TBA

NAME

d2i_DSAPublicKey, i2d_DSAPublicKey, d2i_DSAPrivateKey, i2d_DSAPrivateKey, d2i_DSA_PUBKEY, i2d_DSA_PUBKEY, d2i_DSA_SIG, i2d_DSA_SIG − DSA key encoding and parsing functions.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dsa.h>
#include <openssl/x509.h>

DSA * d2i_DSAPublicKey(DSA **a, const unsigned char **pp, long length);

int i2d_DSAPublicKey(const DSA *a, unsigned char **pp);

DSA * d2i_DSA_PUBKEY(DSA **a, const unsigned char **pp, long length);

int i2d_DSA_PUBKEY(const DSA *a, unsigned char **pp);

DSA * d2i_DSAPrivateKey(DSA **a, const unsigned char **pp, long length);

int i2d_DSAPrivateKey(const DSA *a, unsigned char **pp);

DSA * d2i_DSAparams(DSA **a, const unsigned char **pp, long length);

int i2d_DSAparams(const DSA *a, unsigned char **pp);

DSA * d2i_DSA_SIG(DSA_SIG **a, const unsigned char **pp, long length);

int i2d_DSA_SIG(const DSA_SIG *a, unsigned char **pp);
```

DESCRIPTION

d2i_DSAPublicKey() and *i2d_DSAPublicKey()* decode and encode the DSA public key components structure.

d2i_DSA_PUBKEY() and *i2d_DSA_PUBKEY()* decode and encode an DSA public key using a SubjectPublicKeyInfo (certificate public key) structure.

d2i_DSAPrivateKey(), *i2d_DSAPrivateKey()* decode and encode the DSA private key components.

d2i_DSAparams(), *i2d_DSAparams()* decode and encode the DSA parameters using a **Dss-Parms** structure as defined in RFC2459.

d2i_DSA_SIG(), *i2d_DSA_SIG()* decode and encode a DSA signature using a **Dss-Sig-Value** structure as defined in RFC2459.

The usage of all of these functions is similar to the *d2i_X509()* and *i2d_X509()* described in the *d2i_X509* (3) manual page.

NOTES

The **DSA** structure passed to the private key encoding functions should have all the private key components present.

The data encoded by the private key functions is unencrypted and therefore offers no private key security.

The **DSA_PUBKEY** functions should be used in preference to the **DSAPublicKey** functions when encoding public keys because they use a standard format.

The **DSAPublicKey** functions use an non standard format the actual data encoded depends on the value of the **write_params** field of the **a** key parameter. If **write_params** is zero then only the **pub_key** field is encoded as an **INTEGER**. If **write_params** is 1 then a **SEQUENCE** consisting of the **p**, **q**, **g** and **pub_key** respectively fields are encoded.

The **DSAPrivateKey** functions also use a non standard structure consiting consisting of a SEQUENCE containing the **p**, **q**, **g** and **pub_key** and **priv_key** fields respectively.

SEE ALSO

d2i_X509 (3)

HISTORY

TBA

NAME

d2i_PKCS8PrivateKey_bio, d2i_PKCS8PrivateKey_fp, i2d_PKCS8PrivateKey_bio, i2d_PKCS8PrivateKey_fp, i2d_PKCS8PrivateKey_nid_bio, i2d_PKCS8PrivateKey_nid_fp − PKCS#8 format private key functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>

EVP_PKEY *d2i_PKCS8PrivateKey_bio(BIO *bp, EVP_PKEY **x, pem_password_cb *cb, void
EVP_PKEY *d2i_PKCS8PrivateKey_fp(FILE *fp, EVP_PKEY **x, pem_password_cb *cb, void

int i2d_PKCS8PrivateKey_bio(BIO *bp, EVP_PKEY *x, const EVP_CIPHER *enc,
                            char *kstr, int klen,
                            pem_password_cb *cb, void *u);

int i2d_PKCS8PrivateKey_fp(FILE *fp, EVP_PKEY *x, const EVP_CIPHER *enc,
                           char *kstr, int klen,
                           pem_password_cb *cb, void *u);

int i2d_PKCS8PrivateKey_nid_bio(BIO *bp, EVP_PKEY *x, int nid,
                                char *kstr, int klen,
                                pem_password_cb *cb, void *u);

int i2d_PKCS8PrivateKey_nid_fp(FILE *fp, EVP_PKEY *x, int nid,
                               char *kstr, int klen,
                               pem_password_cb *cb, void *u);
```

DESCRIPTION

The PKCS#8 functions encode and decode private keys in PKCS#8 format using both PKCS#5 v1.5 and PKCS#5 v2.0 password based encryption algorithms.

Other than the use of DER as opposed to PEM these functions are identical to the corresponding **PEM** function as described in the *pem* (3) manual page.

NOTES

Before using these functions *OpenSSL_add_all_algorithms* (3) should be called to initialize the internal algorithm lookup tables otherwise errors about unknown algorithms will occur if an attempt is made to decrypt a private key.

These functions are currently the only way to store encrypted private keys using DER format.

Currently all the functions use BIOs or FILE pointers, there are no functions which work directly on memory: this can be readily worked around by converting the buffers to memory BIOs, see *BIO_s_mem* (3) for details.

SEE ALSO

pem (3)

NAME

d2i_RSAPublicKey, i2d_RSAPublicKey, d2i_RSAPrivateKey, i2d_RSAPrivateKey, d2i_RSA_PUBKEY, i2d_RSA_PUBKEY, i2d_Netscape_RSA, d2i_Netscape_RSA − RSA public and private key encoding functions.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>
#include <openssl/x509.h>

RSA * d2i_RSAPublicKey(RSA **a, unsigned char **pp, long length);

int i2d_RSAPublicKey(RSA *a, unsigned char **pp);

RSA * d2i_RSA_PUBKEY(RSA **a, unsigned char **pp, long length);

int i2d_RSA_PUBKEY(RSA *a, unsigned char **pp);

RSA * d2i_RSAPrivateKey(RSA **a, unsigned char **pp, long length);

int i2d_RSAPrivateKey(RSA *a, unsigned char **pp);

int i2d_Netscape_RSA(RSA *a, unsigned char **pp, int (*cb)());

RSA * d2i_Netscape_RSA(RSA **a, unsigned char **pp, long length, int (*cb)());
```

DESCRIPTION

d2i_RSAPublicKey() and *i2d_RSAPublicKey()* decode and encode a PKCS#1 RSAPublicKey structure.

d2i_RSA_PUBKEY() and *i2d_RSA_PUBKEY()* decode and encode an RSA public key using a SubjectPublicKeyInfo (certificate public key) structure.

d2i_RSAPrivateKey(), *i2d_RSAPrivateKey()* decode and encode a PKCS#1 RSAPrivateKey structure.

d2i_Netscape_RSA(), *i2d_Netscape_RSA()* decode and encode an RSA private key in NET format.

The usage of all of these functions is similar to the *d2i_X509()* and *i2d_X509()* described in the *d2i_X509* (3) manual page.

NOTES

The **RSA** structure passed to the private key encoding functions should have all the PKCS#1 private key components present.

The data encoded by the private key functions is unencrypted and therefore offers no private key security.

The NET format functions are present to provide compatibility with certain very old software. This format has some severe security weaknesses and should be avoided if possible.

SEE ALSO

d2i_X509 (3)

HISTORY

TBA

NAME

d2i_SSL_SESSION, i2d_SSL_SESSION − convert SSL_SESSION object from/to ASN1 representation

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ssl.h>

SSL_SESSION *d2i_SSL_SESSION(SSL_SESSION **a, const unsigned char **pp, long length
int i2d_SSL_SESSION(SSL_SESSION *in, unsigned char **pp);
```

DESCRIPTION

d2i_SSL_SESSION() transforms the external ASN1 representation of an SSL/TLS session, stored as binary data at location **pp** with length **length**, into an SSL_SESSION object.

i2d_SSL_SESSION() transforms the SSL_SESSION object **in** into the ASN1 representation and stores it into the memory location pointed to by **pp**. The length of the resulting ASN1 representation is returned. If **pp** is the NULL pointer, only the length is calculated and returned.

NOTES

The SSL_SESSION object is built from several *malloc()*ed parts, it can therefore not be moved, copied or stored directly. In order to store session data on disk or into a database, it must be transformed into a binary ASN1 representation.

When using *d2i_SSL_SESSION()*, the SSL_SESSION object is automatically allocated. The reference count is 1, so that the session must be explicitly removed using *SSL_SESSION_free* (3), unless the SSL_SESSION object is completely taken over, when being called inside the *get_session_cb()* (see *SSL_CTX_sess_set_get_cb* (3)).

SSL_SESSION objects keep internal link information about the session cache list, when being inserted into one SSL_CTX object's session cache. One SSL_SESSION object, regardless of its reference count, must therefore only be used with one SSL_CTX object (and the SSL objects created from this SSL_CTX object).

When using *i2d_SSL_SESSION()*, the memory location pointed to by **pp** must be large enough to hold the binary representation of the session. There is no known limit on the size of the created ASN1 representation, so the necessary amount of space should be obtained by first calling *i2d_SSL_SESSION()* with **pp=NULL**, and obtain the size needed, then allocate the memory and call *i2d_SSL_SESSION()* again.

RETURN VALUES

d2i_SSL_SESSION() returns a pointer to the newly allocated SSL_SESSION object. In case of failure the NULL-pointer is returned and the error message can be retrieved from the error stack.

i2d_SSL_SESSION() returns the size of the ASN1 representation in bytes. When the session is not valid, **0** is returned and no operation is performed.

SEE ALSO

ssl (3), *SSL_SESSION_free* (3), *SSL_CTX_sess_set_get_cb* (3)

NAME

d2i_X509, i2d_X509, d2i_X509_bio, d2i_X509_fp, i2d_X509_bio, i2d_X509_fp − X509 encode and decode functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>

X509 *d2i_X509(X509 **px, const unsigned char **in, int len);
int i2d_X509(X509 *x, unsigned char **out);

X509 *d2i_X509_bio(BIO *bp, X509 **x);
X509 *d2i_X509_fp(FILE *fp, X509 **x);

int i2d_X509_bio(X509 *x, BIO *bp);
int i2d_X509_fp(X509 *x, FILE *fp);
```

DESCRIPTION

The X509 encode and decode routines encode and parse an **X509** structure, which represents an X509 certificate.

d2i_X509() attempts to decode **len** bytes at ***in**. If successful a pointer to the **X509** structure is returned. If an error occurred then **NULL** is returned. If **px** is not **NULL** then the returned structure is written to ***px**. If ***px** is not **NULL** then it is assumed that ***px** contains a valid **X509** structure and an attempt is made to reuse it. If the call is successful ***in** is incremented to the byte following the parsed data.

i2d_X509() encodes the structure pointed to by **x** into DER format. If **out** is not **NULL** is writes the DER encoded data to the buffer at ***out**, and increments it to point after the data just written. If the return value is negative an error occurred, otherwise it returns the length of the encoded data.

For OpenSSL 0.9.7 and later if ***out** is **NULL** memory will be allocated for a buffer and the encoded data written to it. In this case ***out** is not incremented and it points to the start of the data just written.

d2i_X509_bio() is similar to *d2i_X509()* except it attempts to parse data from BIO **bp**.

d2i_X509_fp() is similar to *d2i_X509()* except it attempts to parse data from FILE pointer **fp**.

i2d_X509_bio() is similar to *i2d_X509()* except it writes the encoding of the structure **x** to BIO **bp** and it returns 1 for success and 0 for failure.

i2d_X509_fp() is similar to *i2d_X509()* except it writes the encoding of the structure **x** to BIO **bp** and it returns 1 for success and 0 for failure.

NOTES

The letters **i** and **d** in for example **i2d_X509** stand for ''internal'' (that is an internal C structure) and ''DER''. So that **i2d_X509** converts from internal to DER.

The functions can also understand **BER** forms.

The actual X509 structure passed to *i2d_X509()* must be a valid populated **X509** structure it can **not** simply be fed with an empty structure such as that returned by *X509_new()*.

The encoded data is in binary form and may contain embedded zeroes. Therefore any FILE pointers or BIOs should be opened in binary mode. Functions such as ***strlen()*** will **not** return the correct length of the encoded structure.

The ways that ***in** and ***out** are incremented after the operation can trap the unwary. See the **WARNINGS** section for some common errors.

The reason for the auto increment behaviour is to reflect a typical usage of ASN1 functions: after one structure is encoded or decoded another will processed after it.

EXAMPLES

Allocate and encode the DER encoding of an X509 structure:

```
int len;
unsigned char *buf, *p;

len = i2d_X509(x, NULL);

buf = OPENSSL_malloc(len);

if (buf == NULL)
       /* error */

p = buf;

i2d_X509(x, &p);
```

If you are using OpenSSL 0.9.7 or later then this can be simplified to:

```
int len;
unsigned char *buf;

buf = NULL;

len = i2d_X509(x, &buf);

if (len < 0)
       /* error */
```

Attempt to decode a buffer:

```
X509 *x;

unsigned char *buf, *p;

int len;

/* Something to setup buf and len */

p = buf;

x = d2i_X509(NULL, &p, len);

if (x == NULL)
   /* Some error */
```

Alternative technique:

```
X509 *x;

unsigned char *buf, *p;

int len;

/* Something to setup buf and len */

p = buf;

x = NULL;

if(!d2i_X509(&x, &p, len))
   /* Some error */
```

WARNINGS

The use of temporary variable is mandatory. A common mistake is to attempt to use a buffer directly as follows:

```
int len;
unsigned char *buf;

len = i2d_X509(x, NULL);
```

```
buf = OPENSSL_malloc(len);
if (buf == NULL)
       /* error */
i2d_X509(x, &buf);
/* Other stuff ... */
OPENSSL_free(buf);
```

This code will result in **buf** apparently containing garbage because it was incremented after the call to point after the data just written. Also **buf** will no longer contain the pointer allocated by ***OPENSSL_malloc()*** and the subsequent call to ***OPENSSL_free()*** may well crash.

The auto allocation feature (setting buf to NULL) only works on OpenSSL 0.9.7 and later. Attempts to use it on earlier versions will typically cause a segmentation violation.

Another trap to avoid is misuse of the **xp** argument to ***d2i_X509()***:

```
X509 *x;
if (!d2i_X509(&x, &p, len))
       /* Some error */
```

This will probably crash somewhere in ***d2i_X509()***. The reason for this is that the variable **x** is uninitialized and an attempt will be made to interpret its (invalid) value as an **X509** structure, typically causing a segmentation violation. If **x** is set to NULL first then this will not happen.

BUGS

In some versions of OpenSSL the ''reuse'' behaviour of *d2i_X509()* when ***px** is valid is broken and some parts of the reused structure may persist if they are not present in the new one. As a result the use of this ''reuse'' behaviour is strongly discouraged.

i2d_X509() will not return an error in many versions of OpenSSL, if mandatory fields are not initialized due to a programming error then the encoded structure may contain invalid data or omit the fields entirely and will not be parsed by *d2i_X509()*. This may be fixed in future so code should not assume that *i2d_X509()* will always succeed.

RETURN VALUES

d2i_X509(), *d2i_X509_bio()* and *d2i_X509_fp()* return a valid **X509** structure or **NULL** if an error occurs. The error code that can be obtained by *ERR_get_error* (3).

i2d_X509(), *i2d_X509_bio()* and *i2d_X509_fp()* return a the number of bytes successfully encoded or a negative value if an error occurs. The error code can be obtained by *ERR_get_error* (3).

i2d_X509_bio() and *i2d_X509_fp()* returns 1 for success and 0 if an error occurs The error code can be obtained by *ERR_get_error* (3).

SEE ALSO

ERR_get_error (3)

HISTORY

d2i_X509, i2d_X509, d2i_X509_bio, d2i_X509_fp, i2d_X509_bio and i2d_X509_fp are available in all versions of SSLeay and OpenSSL.

NAME

d2i_X509_ALGOR, i2d_X509_ALGOR − AlgorithmIdentifier functions.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>

X509_ALGOR *d2i_X509_ALGOR(X509_ALGOR **a, unsigned char **pp, long length);
int i2d_X509_ALGOR(X509_ALGOR *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an **X509_ALGOR** structure which is equivalent to the **AlgorithmIdentifier** structure.

Othewise these behave in a similar way to *d2i_X509()* and *i2d_X509()* described in the *d2i_X509* (3) manual page.

SEE ALSO

d2i_X509 (3)

HISTORY

TBA

NAME

d2i_X509_CRL, i2d_X509_CRL, d2i_X509_CRL_bio, d2i_509_CRL_fp, i2d_X509_CRL_bio, i2d_X509_CRL_fp − PKCS#10 certificate request functions.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>

X509_CRL *d2i_X509_CRL(X509_CRL **a, const unsigned char **pp, long length);
int i2d_X509_CRL(X509_CRL *a, unsigned char **pp);

X509_CRL *d2i_X509_CRL_bio(BIO *bp, X509_CRL **x);
X509_CRL *d2i_X509_CRL_fp(FILE *fp, X509_CRL **x);

int i2d_X509_CRL_bio(X509_CRL *x, BIO *bp);
int i2d_X509_CRL_fp(X509_CRL *x, FILE *fp);
```

DESCRIPTION

These functions decode and encode an X509 CRL (certificate revocation list).

Othewise the functions behave in a similar way to *d2i_X509()* and *i2d_X509()* described in the *d2i_X509* (3) manual page.

SEE ALSO

d2i_X509 (3)

HISTORY

TBA

NAME

d2i_X509_NAME, i2d_X509_NAME − X509_NAME encoding functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>

X509_NAME *d2i_X509_NAME(X509_NAME **a, unsigned char **pp, long length);
int i2d_X509_NAME(X509_NAME *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an **X509_NAME** structure which is the the same as the **Name** type defined in RFC2459 (and elsewhere) and used for example in certificate subject and issuer names.

Othewise the functions behave in a similar way to *d2i_X509()* and *i2d_X509()* described in the *d2i_X509* (3) manual page.

SEE ALSO

d2i_X509 (3)

HISTORY

TBA

NAME

d2i_X509_REQ, i2d_X509_REQ, d2i_X509_REQ_bio, d2i_X509_REQ_fp, i2d_X509_REQ_bio, i2d_X509_REQ_fp − PKCS#10 certificate request functions.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>

X509_REQ *d2i_X509_REQ(X509_REQ **a, const unsigned char **pp, long length);
int i2d_X509_REQ(X509_REQ *a, unsigned char **pp);

X509_REQ *d2i_X509_REQ_bio(BIO *bp, X509_REQ **x);
X509_REQ *d2i_X509_REQ_fp(FILE *fp, X509_REQ **x);

int i2d_X509_REQ_bio(X509_REQ *x, BIO *bp);
int i2d_X509_REQ_fp(X509_REQ *x, FILE *fp);
```

DESCRIPTION

These functions decode and encode a PKCS#10 certificate request.

Othewise these behave in a similar way to *d2i_X509()* and *i2d_X509()* described in the *d2i_X509* (3) manual page.

SEE ALSO

d2i_X509 (3)

HISTORY

TBA

NAME

d2i_X509_SIG, i2d_X509_SIG − DigestInfo functions.

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>

X509_SIG *d2i_X509_SIG(X509_SIG **a, unsigned char **pp, long length);
int i2d_X509_SIG(X509_SIG *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an X509_SIG structure which is equivalent to the **DigestInfo** structure defined in PKCS#1 and PKCS#7.

Othewise these behave in a similar way to *d2i_X509()* and *i2d_X509()* described in the *d2i_X509* (3) manual page.

SEE ALSO

d2i_X509 (3)

HISTORY

TBA

NAME

daemon — run in the background

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
daemon(int nochdir, int noclose);
```

DESCRIPTION

The **daemon**() function is for programs wishing to detach themselves from the controlling terminal and run in the background as system daemons.

Unless the argument *nochdir* is non-zero, **daemon**() changes the current working directory to the root (`/`).

Unless the argument *noclose* is non-zero, **daemon**() will redirect standard input, standard output and standard error to `/dev/null`.

RETURN VALUES

On return 0 indicates success with −1 indicating error.

ERRORS

The function **daemon**() may fail and set *errno* for any of the errors specified for the library functions `fork`(2) and `setsid`(2).

SEE ALSO

`fork`(2), `setsid`(2)

HISTORY

The **daemon**() function first appeared in 4.4BSD.

CAVEATS

Unless the *noclose* argument is non-zero, **daemon**() will close the first three file descriptors and redirect them to `/dev/null`. Normally, these correspond to standard input, standard output and standard error. However, if any of those file descriptors refer to something else they will still be closed, resulting in incorrect behavior of the calling program. This can happen if any of standard input, standard output or standard error have been closed before the program was run. Programs using **daemon**() should therefore make sure to either call **daemon**() before opening any files or sockets or, alternately, verifying that any file descriptors obtained have a value greater than 2.

BUGS

daemon() uses **fork**() as part of its tty detachment mechanism. Consequently the process id changes when **daemon**() is invoked. Processes employing **daemon**() can not be reliably waited upon until **daemon**() has been invoked.

NAME

dbm_clearerr, **dbm_close**, **dbm_delete**, **dbm_dirfno**, **dbm_error**, **dbm_fetch**, **dbm_firstkey**, **dbm_nextkey**, **dbm_open**, **dbm_store**, **ndbm** — database functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ndbm.h>

int
dbm_clearerr(DBM *db);

void
dbm_close(DBM *db);

int
dbm_delete(DBM *db, datum key);

int
dbm_dirfno(DBM *db);

int
dbm_error(DBM *db);

datum
dbm_fetch(DBM *db, datum key);

datum
dbm_firstkey(DBM *db);

datum
dbm_nextkey(DBM *db);

DBM *
dbm_open(const char *file, int open_flags, mode_t file_mode);

int
dbm_store(DBM *db, datum key, datum content, int store_mode);
```

DESCRIPTION

The **ndbm** facility provides access to hash database files.

Two data types are fundamental to the **ndbm** facility. *DBM* serves as a handle to a database. It is an opaque type.

The other data type is *datum*, which is a structure type which includes the following members:

```
void *  dptr
size_t  dsize
```

A *datum* is thus given by *dptr* pointing at an object of *dsize* bytes in length.

The **dbm_open**() function opens a database. The *file* argument is the pathname which the actual database file pathname is based on. This implementation uses a single file with the suffix `.db` appended to *file*. The *open_flags* argument has the same meaning as the *flags* argument to open(2) except that when opening a database for write-only access the file is opened for read/write access, and the `O_APPEND` flag must not be specified. The *file_mode* argument has the same meaning as the *mode* argument to open(2).

For the following functions, the *db* argument is a handle previously returned by a call to **dbm_open**().

The **dbm_close**() function closes a database.

The **dbm_fetch**() function retrieves a record from the database. The *key* argument is a *datum* that identifies the record to be fetched.

The **dbm_store**() function stores a record into the database. The *key* argument is a *datum* that identifies the record to be stored. The *content* argument is a *datum* that specifies the value of the record to be stored. The *store_mode* argument specifies the behavior of **dbm_store**() if a record matching *key* is already present in the database, *db*. *store_mode* must be one of the following:

DBM_INSERT If a record matching *key* is already present, it is left unchanged.

DBM_REPLACE If a record matching *key* is already present, its value is replaced by *content*.

If no record matching *key* is present, a new record is inserted regardless of *store_mode*.

The **dbm_delete**() function deletes a record from the database. The *key* argument is a *datum* that identifies the record to be deleted.

The **dbm_firstkey**() function returns the first key in the database.

The **dbm_nextkey**() function returns the next key in the database. In order to be meaningful, it must be preceded by a call to **dbm_firstkey**().

The **dbm_error**() function returns the error indicator of the database.

The **dbm_clearerr**() function clears the error indicator of the database.

The **dbm_dirfno**() function returns the file descriptor of the underlying database file.

IMPLEMENTATION NOTES

The **ndbm** facility is implemented on top of the hash(3) access method of the db(3) database facility.

RETURN VALUES

The **dbm_open**() function returns a pointer to a *DBM* when successful; otherwise a null pointer is returned.

The **dbm_close**() function returns no value.

The **dbm_fetch**() function returns a content *datum*; if no record matching *key* was found or if an error occured, its *dptr* member is a null pointer.

The **dbm_store**() function returns 0 when then record was successfully inserted; it returns 1 when called with *store_mode* being DBM_INSERT and a record matching *key* is already present; otherwise a negative value is returned.

The **dbm_delete**() function returns 0 when the record was successfully deleted; otherwise a negative value is returned.

The **dbm_firstkey**() and **dbm_nextkey**() functions return a key *datum*. When the end of the database is reached or if an error occured, its *dptr* member is a null pointer.

The **dbm_error**() function returns 0 if the error indicator is clear; if the error indicator is set a non-zero value is returned.

The **dbm_clearerr**() function always returns 0.

The **dbm_dirfno**() function returns the file descriptor of the underlying database file.

ERRORS

No errors are defined.

SEE ALSO

open(2), db(3), hash(3)

STANDARDS

The **dbm_clearerr**(), **dbm_close**(), **dbm_delete**(), **dbm_error**(), **dbm_fetch**(), **dbm_firstkey**(), **dbm_nextkey**(), **dbm_open**(), and **dbm_store**() functions conform to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2"). The **dbm_dirfno**() function is an extension.

NAME

dbopen, **db** — database access methods

SYNOPSIS

```
#include <sys/types.h>
#include <limits.h>
#include <db.h>
#include <fcntl.h>

DB *
dbopen(const char *file, int flags, mode_t mode, DBTYPE type,
    const void *openinfo);
```

DESCRIPTION

dbopen is the library interface to database files. The supported file formats are btree, hashed, and UNIX file oriented. The btree format is a representation of a sorted, balanced tree structure. The hashed format is an extensible, dynamic hashing scheme. The flat-file format is a byte stream file with fixed or variable length records. The formats and file format specific information are described in detail in their respective manual pages btree(3), hash(3), and recno(3).

dbopen opens *file* for reading and/or writing. Files never intended to be preserved on disk may be created by setting the file parameter to NULL.

The *flags* and *mode* arguments are as specified to the open(2) routine, however, only the O_CREAT, O_EXCL, O_EXLOCK, O_NONBLOCK, O_RDONLY, O_RDWR, O_SHLOCK, and O_TRUNC flags are meaningful. (Note, opening a database file O_WRONLY is not possible.)

The *type* argument is of type *DBTYPE* (as defined in the ⟨db.h⟩ include file) and may be set to DB_BTREE, DB_HASH, or DB_RECNO.

The *openinfo* argument is a pointer to an access method specific structure described in the access method's manual page. If *openinfo* is NULL, each access method will use defaults appropriate for the system and the access method.

dbopen returns a pointer to a DB structure on success and NULL on error. The DB structure is defined in the ⟨db.h⟩ include file, and contains at least the following fields:

```
typedef struct {
        DBTYPE type;
        int (*close)(const DB *db);
        int (*del)(const DB *db, const DBT *key, u_int flags);
        int (*fd)(const DB *db);
        int (*get)(const DB *db, DBT *key, DBT *data, u_int flags);
        int (*put)(const DB *db, DBT *key, const DBT *data,
            u_int flags);
        int (*sync)(const DB *db, u_int flags);
        int (*seq)(const DB *db, DBT *key, DBT *data, u_int flags);
} DB;
```

These elements describe a database type and a set of functions performing various actions. These functions take a pointer to a structure as returned by **dbopen**, and sometimes one or more pointers to key/data structures and a flag value.

type The type of the underlying access method (and file format).

close A pointer to a routine to flush any cached information to disk, free any allocated resources, and close the underlying file(s). Since key/data pairs may be cached in memory, failing to sync the file with a *close* or *sync* function may result in inconsistent or lost information. *close* routines return −1 on error (setting *errno*) and 0 on success.

del A pointer to a routine to remove key/data pairs from the database.

The parameter *flag* may be set to the following value:

R_CURSOR Delete the record referenced by the cursor. The cursor must have previously been initialized.

delete routines return −1 on error (setting *errno*), 0 on success, and 1 if the specified *key* was not in the file.

fd A pointer to a routine which returns a file descriptor representative of the underlying database. A file descriptor referencing the same file will be returned to all processes which call **dbopen** with the same *file* name. This file descriptor may be safely used as an argument to the fcntl(2) and flock(2) locking functions. The file descriptor is not necessarily associated with any of the underlying files used by the access method. No file descriptor is available for in memory databases. *fd* routines return −1 on error (setting *errno*), and the file descriptor on success.

get A pointer to a routine which is the interface for keyed retrieval from the database. The address and length of the data associated with the specified *key* are returned in the structure referenced by *data*. *get* routines return −1 on error (setting *errno*), 0 on success, and 1 if the *key* was not in the file.

put A pointer to a routine to store key/data pairs in the database.

The parameter *flag* may be set to one of the following values:

R_CURSOR Replace the key/data pair referenced by the cursor. The cursor must have previously been initialized.

R_IAFTER Append the data immediately after the data referenced by *key*, creating a new key/data pair. The record number of the appended key/data pair is returned in the *key* structure. (Applicable only to the DB_RECNO access method.)

R_IBEFORE Insert the data immediately before the data referenced by *key*, creating a new key/data pair. The record number of the inserted key/data pair is returned in the *key* structure. (Applicable only to the DB_RECNO access method.)

R_NOOVERWRITE Enter the new key/data pair only if the key does not previously exist.

R_SETCURSOR Store the key/data pair, setting or initializing the position of the cursor to reference it. (Applicable only to the DB_BTREE and DB_RECNO access methods.)

R_SETCURSOR is available only for the DB_BTREE and DB_RECNO access methods because it implies that the keys have an inherent order which does not change.

R_IAFTER and R_IBEFORE are available only for the DB_RECNO access method because they each imply that the access method is able to create new keys. This is only true if the keys are ordered and independent, record numbers for example.

The default behavior of the *put* routines is to enter the new key/data pair, replacing any previously existing key.

put routines return −1 on error (setting *errno*), 0 on success, and 1 if the R_NOOVERWRITE *flag* was set and the key already exists in the file.

seq A pointer to a routine which is the interface for sequential retrieval from the database. The address and length of the key are returned in the structure referenced by *key*, and the address and length of the data are returned in the structure referenced by *data*.

Sequential key/data pair retrieval may begin at any time, and the position of the "cursor" is not affected by calls to the *del*, *get*, *put*, or *sync* routines. Modifications to the database during a sequential scan will be reflected in the scan, i.e., records inserted behind the cursor will not be returned while records inserted in front of the cursor will be returned.

The flag value *must* be set to one of the following values:

R_CURSOR The data associated with the specified key is returned. This differs from the *get* routines in that it sets or initializes the cursor to the location of the key as well. (Note, for the DB_BTREE access method, the returned key is not necessarily an exact match for the specified key. The returned key is the smallest key greater than or equal to the specified key, permitting partial key matches and range searches.)

R_FIRST The first key/data pair of the database is returned, and the cursor is set or initialized to reference it.

R_LAST The last key/data pair of the database is returned, and the cursor is set or initialized to reference it. (Applicable only to the DB_BTREE and DB_RECNO access methods.)

R_NEXT Retrieve the key/data pair immediately after the cursor. If the cursor is not yet set, this is the same as the R_FIRST flag.

R_PREV Retrieve the key/data pair immediately before the cursor. If the cursor is not yet set, this is the same as the R_LAST flag. (Applicable only to the DB_BTREE and DB_RECNO access methods.)

R_LAST and R_PREV are available only for the DB_BTREE and DB_RECNO access methods because they each imply that the keys have an inherent order which does not change.

seq routines return −1 on error (setting *errno*), 0 on success and 1 if there are no key/data pairs less than or greater than the specified or current key. If the DB_RECNO access method is being used, and if the database file is a character special file and no complete key/data pairs are currently available, the *seq* routines return 2.

sync A pointer to a routine to flush any cached information to disk. If the database is in memory only, the *sync* routine has no effect and will always succeed.

The flag value may be set to the following value:

R_RECNOSYNC
If the DB_RECNO access method is being used, this flag causes the sync routine to apply to the btree file which underlies the recno file, not the recno file itself. (See the *bfname* field of the recno(3) manual page for more information.)

sync routines return −1 on error (setting *errno*) and 0 on success.

KEY/DATA PAIRS

Access to all file types is based on key/data pairs. Both keys and data are represented by the following data structure:

```
typedef struct {
        void *data;
        size_t size;
} DBT;
```

The elements of the DBT structure are defined as follows:

data A pointer to a byte string.

size The length of the byte string.

Key and data byte strings may reference strings of essentially unlimited length although any two of them must fit into available memory at the same time. It should be noted that the access methods provide no guarantees about byte string alignment.

ERRORS

The **dbopen** routine may fail and set *errno* for any of the errors specified for the library routines open(2) and malloc(3) or the following:

EFTYPE A file is incorrectly formatted.

EINVAL A parameter has been specified (hash function, pad byte, etc.) that is incompatible with the current file specification or which is not meaningful for the function (for example, use of the cursor without prior initialization) or there is a mismatch between the version number of file and the software.

EFBIG The key could not be inserted due to limitations in the DB file format (e.g., a hash database was out of overflow pages).

The *close* routines may fail and set *errno* for any of the errors specified for the library routines close(2), read(2), write(2), free(3), or fsync(2).

The *del*, *get*, *put*, and *seq* routines may fail and set *errno* for any of the errors specified for the library routines read(2), write(2), free(3), or malloc(3).

The *fd* routines will fail and set *errno* to ENOENT for in memory databases.

The *sync* routines may fail and set *errno* for any of the errors specified for the library routine fsync(2).

SEE ALSO

btree(3), hash(3), mpool(3), recno(3)

Margo Seltzer and Michael Olson, "LIBTP: Portable, Modular Transactions for UNIX", *USENIX proceedings*, Winter 1992.

BUGS

The typedef DBT is a mnemonic for “data base thang”, and was used because no one could think of a reasonable name that wasn’t already used.

The file descriptor interface is a kludge and will be deleted in a future version of the interface.

None of the access methods provide any form of concurrent access, locking, or transactions.

NAME

des_random_key, des_set_key, des_key_sched, des_set_key_checked, des_set_key_unchecked, des_set_odd_parity, des_is_weak_key, des_ecb_encrypt, des_ecb2_encrypt, des_ecb3_encrypt, des_ncbc_encrypt, des_cfb_encrypt, des_ofb_encrypt, des_pcbc_encrypt, des_cfb64_encrypt, des_ofb64_encrypt, des_xcbc_encrypt, des_ede2_cbc_encrypt, des_ede2_cfb64_encrypt, des_ede2_ofb64_encrypt, des_ede3_cbc_encrypt, des_ede3_cbcm_encrypt, des_ede3_cfb64_encrypt, des_ede3_ofb64_encrypt, des_read_password, des_read_2passwords, des_read_pw_string, des_cbc_cksum, des_quad_cksum, des_string_to_key, des_string_to_2keys, des_fcrypt, des_crypt, des_enc_read, des_enc_write − DES encryption

SYNOPSIS

```
#include <openssl/des.h>

void des_random_key(des_cblock *ret);

int des_set_key(const_des_cblock *key, des_key_schedule schedule);
int des_key_sched(const_des_cblock *key, des_key_schedule schedule);
int des_set_key_checked(const_des_cblock *key,
       des_key_schedule schedule);
void des_set_key_unchecked(const_des_cblock *key,
       des_key_schedule schedule);

void des_set_odd_parity(des_cblock *key);
int des_is_weak_key(const_des_cblock *key);

void des_ecb_encrypt(const_des_cblock *input, des_cblock *output,
       des_key_schedule ks, int enc);
void des_ecb2_encrypt(const_des_cblock *input, des_cblock *output,
       des_key_schedule ks1, des_key_schedule ks2, int enc);
void des_ecb3_encrypt(const_des_cblock *input, des_cblock *output,
       des_key_schedule ks1, des_key_schedule ks2,
       des_key_schedule ks3, int enc);

void des_ncbc_encrypt(const unsigned char *input, unsigned char *output,
       long length, des_key_schedule schedule, des_cblock *ivec,
       int enc);
void des_cfb_encrypt(const unsigned char *in, unsigned char *out,
       int numbits, long length, des_key_schedule schedule,
       des_cblock *ivec, int enc);
void des_ofb_encrypt(const unsigned char *in, unsigned char *out,
       int numbits, long length, des_key_schedule schedule,
       des_cblock *ivec);
void des_pcbc_encrypt(const unsigned char *input, unsigned char *output,
       long length, des_key_schedule schedule, des_cblock *ivec,
       int enc);
void des_cfb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, des_key_schedule schedule, des_cblock *ivec,
       int *num, int enc);
void des_ofb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, des_key_schedule schedule, des_cblock *ivec,
       int *num);
```

```
void des_xcbc_encrypt(const unsigned char *input, unsigned char *output,
       long length, des_key_schedule schedule, des_cblock *ivec,
       const_des_cblock *inw, const_des_cblock *outw, int enc);

void des_ede2_cbc_encrypt(const unsigned char *input,
       unsigned char *output, long length, des_key_schedule ks1,
       des_key_schedule ks2, des_cblock *ivec, int enc);
void des_ede2_cfb64_encrypt(const unsigned char *in,
       unsigned char *out, long length, des_key_schedule ks1,
       des_key_schedule ks2, des_cblock *ivec, int *num, int enc);
void des_ede2_ofb64_encrypt(const unsigned char *in,
       unsigned char *out, long length, des_key_schedule ks1,
       des_key_schedule ks2, des_cblock *ivec, int *num);

void des_ede3_cbc_encrypt(const unsigned char *input,
       unsigned char *output, long length, des_key_schedule ks1,
       des_key_schedule ks2, des_key_schedule ks3, des_cblock *ivec,
       int enc);
void des_ede3_cbcm_encrypt(const unsigned char *in, unsigned char *out,
       long length, des_key_schedule ks1, des_key_schedule ks2,
       des_key_schedule ks3, des_cblock *ivec1, des_cblock *ivec2,
       int enc);
void des_ede3_cfb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, des_key_schedule ks1, des_key_schedule ks2,
       des_key_schedule ks3, des_cblock *ivec, int *num, int enc);
void des_ede3_ofb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, des_key_schedule ks1,
       des_key_schedule ks2, des_key_schedule ks3,
       des_cblock *ivec, int *num);

int des_read_password(des_cblock *key, const char *prompt, int verify);
int des_read_2passwords(des_cblock *key1, des_cblock *key2,
       const char *prompt, int verify);
int des_read_pw_string(char *buf, int length, const char *prompt,
       int verify);

DES_LONG des_cbc_cksum(const unsigned char *input, des_cblock *output,
       long length, des_key_schedule schedule,
       const_des_cblock *ivec);
DES_LONG des_quad_cksum(const unsigned char *input, des_cblock output[],
       long length, int out_count, des_cblock *seed);
void des_string_to_key(const char *str, des_cblock *key);
void des_string_to_2keys(const char *str, des_cblock *key1,
       des_cblock *key2);

char *des_fcrypt(const char *buf, const char *salt, char *ret);
char *des_crypt(const char *buf, const char *salt);
char *crypt(const char *buf, const char *salt);

int des_enc_read(int fd, void *buf, int len, des_key_schedule sched,
       des_cblock *iv);
int des_enc_write(int fd, const void *buf, int len,
       des_key_schedule sched, des_cblock *iv);
```

DESCRIPTION

This library contains a fast implementation of the DES encryption algorithm.

There are two phases to the use of DES encryption. The first is the generation of a *des_key_schedule* from a key, the second is the actual encryption. A DES key is of type *des_cblock*. This type is consists of 8 bytes with odd parity. The least significant bit in each byte is the parity bit. The key schedule is an expanded form of the key; it is used to speed the encryption process.

des_random_key() generates a random key. The PRNG must be seeded prior to using this function (see rand(3); for backward compatibility the function *des_random_seed()* is available as well). If the PRNG could not generate a secure key, 0 is returned. In earlier versions of the library, *des_random_key()* did not generate secure keys.

Before a DES key can be used, it must be converted into the architecture dependent *des_key_schedule* via the *des_set_key_checked()* or *des_set_key_unchecked()* function.

des_set_key_checked() will check that the key passed is of odd parity and is not a week or semi-weak key. If the parity is wrong, then −1 is returned. If the key is a weak key, then −2 is returned. If an error is returned, the key schedule is not generated.

des_set_key() (called *des_key_sched()* in the MIT library) works like *des_set_key_checked()* if the *des_check_key* flag is non-zero, otherwise like *des_set_key_unchecked()*. These functions are available for compatibility; it is recommended to use a function that does not depend on a global variable.

des_set_odd_parity() (called *des_fixup_key_parity()* in the MIT library) sets the parity of the passed *key* to odd.

des_is_weak_key() returns 1 is the passed key is a weak key, 0 if it is ok. The probability that a randomly generated key is weak is 1/2^52, so it is not really worth checking for them.

The following routines mostly operate on an input and output stream of *des_cblock*s.

des_ecb_encrypt() is the basic DES encryption routine that encrypts or decrypts a single 8−byte *des_cblock* in *electronic code book* (ECB) mode. It always transforms the input data, pointed to by *input*, into the output data, pointed to by the *output* argument. If the *encrypt* argument is non-zero (DES_ENCRYPT), the *input* (cleartext) is encrypted in to the *output* (ciphertext) using the key_schedule specified by the *schedule* argument, previously set via *des_set_key*. If *encrypt* is zero (DES_DECRYPT), the *input* (now ciphertext) is decrypted into the *output* (now cleartext). Input and output may overlap. *des_ecb_encrypt()* does not return a value.

des_ecb3_encrypt() encrypts/decrypts the *input* block by using three-key Triple-DES encryption in ECB mode. This involves encrypting the input with *ks1*, decrypting with the key schedule *ks2*, and then encrypting with *ks3*. This routine greatly reduces the chances of brute force breaking of DES and has the advantage of if *ks1*, *ks2* and *ks3* are the same, it is equivalent to just encryption using ECB mode and *ks1* as the key.

The macro *des_ecb2_encrypt()* is provided to perform two-key Triple-DES encryption by using *ks1* for the final encryption.

des_ncbc_encrypt() encrypts/decrypts using the *cipher-block-chaining* (CBC) mode of DES. If the *encrypt* argument is non-zero, the routine cipher-block-chain encrypts the cleartext data pointed to by the *input* argument into the ciphertext pointed to by the *output* argument, using the key schedule provided by the *schedule* argument, and initialization vector provided by the *ivec* argument. If the *length* argument is not an integral multiple of eight bytes, the last block is copied to a temporary area and zero filled. The output is always an integral multiple of eight bytes.

des_xcbc_encrypt() is RSA's DESX mode of DES. It uses *inw* and *outw* to 'whiten' the encryption. *inw* and *outw* are secret (unlike the iv) and are as such, part of the key. So the key is sort of 24 bytes. This is much better than CBC DES.

des_ede3_cbc_encrypt() implements outer triple CBC DES encryption with three keys. This means that each DES operation inside the CBC mode is really an `C=E(ks3,D(ks2,E(ks1,M)))`. This mode is used by SSL.

The *des_ede2_cbc_encrypt()* macro implements two-key Triple-DES by reusing *ks1* for the final encryption. `C=E(ks1,D(ks2,E(ks1,M)))`. This form of Triple-DES is used by the RSAREF library.

des_cfb_encrypt() encrypt/decrypts using cipher feedback mode. This method takes an array of characters as input and outputs and array of characters. It does not require any padding to 8 character groups. Note: the *ivec* variable is changed and the new changed value needs to be passed to the next call to this function. Since this function runs a complete DES ECB encryption per *numbits*, this function is only suggested for use when sending small numbers of characters.

des_cfb64_encrypt() implements CFB mode of DES with 64bit feedback. Why is this useful you ask? Because this routine will allow you to encrypt an arbitrary number of bytes, no 8 byte padding. Each call to this routine will encrypt the input bytes to output and then update ivec and num. num contains 'how far' we are though ivec. If this does not make much sense, read more about cfb mode of DES :−).

des_ede3_cfb64_encrypt() and *des_ede2_cfb64_encrypt()* is the same as *des_cfb64_encrypt()* except that Triple-DES is used.

des_ofb_encrypt() encrypts using output feedback mode. This method takes an array of characters as input and outputs and array of characters. It does not require any padding to 8 character groups. Note: the *ivec* variable is changed and the new changed value needs to be passed to the next call to this function. Since this function runs a complete DES ECB encryption per numbits, this function is only suggested for use when sending small numbers of characters.

des_ofb64_encrypt() is the same as *des_cfb64_encrypt()* using Output Feed Back mode.

des_ede3_ofb64_encrypt() and *des_ede2_ofb64_encrypt()* is the same as *des_ofb64_encrypt()*, using Triple-DES.

des_read_pw_string() writes the string specified by *prompt* to standard output, turns echo off and reads in input string from the terminal. The string is returned in *buf*, which must have space for at least *length* bytes. If *verify* is set, the user is asked for the password twice and unless the two copies match, an error is returned. A return code of −1 indicates a system error, 1 failure due to use interaction, and 0 is success.

des_read_password() does the same and converts the password to a DES key by calling *des_string_to_key()*; *des_read_2password()* operates in the same way as *des_read_password()* except that it generates two keys by using the *des_string_to_2key()* function. *des_string_to_key()* is available for backward compatibility with the MIT library. New applications should use a cryptographic hash function. The same applies for *des_string_to_2key()*.

The following are DES-based transformations:

des_fcrypt() is a fast version of the Unix *crypt*(3) function. This version takes only a small amount of space relative to other fast *crypt()* implementations. This is different to the normal crypt in that the third parameter is the buffer that the return value is written into. It needs to be at least 14 bytes long. This function is thread safe, unlike the normal crypt.

des_crypt() is a faster replacement for the normal system *crypt()*. This function calls *des_fcrypt()* with a static array passed as the third parameter. This emulates the normal non-thread safe semantics of *crypt*(3).

des_enc_write() writes *len* bytes to file descriptor *fd* from buffer *buf*. The data is encrypted via *pcbc_encrypt* (default) using *sched* for the key and *iv* as a starting vector. The actual data send down *fd* consists of 4 bytes (in network byte order) containing the length of the following encrypted data. The encrypted data then follows, padded with random data out to a multiple of 8 bytes.

des_enc_read() is used to read *len* bytes from file descriptor *fd* into buffer *buf*. The data being read from *fd* is assumed to have come from *des_enc_write()* and is decrypted using *sched* for the key schedule and *iv* for the initial vector.

Warning: The data format used by *des_enc_write()* and *des_enc_read()* has a cryptographic weakness: When asked to write more than MAXWRITE bytes, *des_enc_write()* will split the data into several chunks that are all encrypted using the same IV. So don't use these functions unless you are sure you know what you do (in which case you might not want to use them anyway). They cannot handle non-blocking sockets.

des_enc_read() uses an internal state and thus cannot be used on multiple files.

des_rw_mode is used to specify the encryption mode to use with *des_enc_read()* and *des_end_write()*. If set to *DES_PCBC_MODE* (the default), des_pcbc_encrypt is used. If set to *DES_CBC_MODE* des_cbc_encrypt is used.

NOTES

Single-key DES is insecure due to its short key size. ECB mode is not suitable for most applications; see des_modes(7).

The evp(3) library provides higher-level encryption functions.

BUGS

des_3cbc_encrypt() is flawed and must not be used in applications.

des_cbc_encrypt() does not modify **ivec**; use *des_ncbc_encrypt()* instead.

des_cfb_encrypt() and *des_ofb_encrypt()* operates on input of 8 bits. What this means is that if you set numbits to 12, and length to 2, the first 12 bits will come from the 1st input byte and the low half of the second input byte. The second 12 bits will have the low 8 bits taken from the 3rd input byte and the top 4 bits taken from the 4th input byte. The same holds for output. This function has been implemented this way because most people will be using a multiple of 8 and because once you get into pulling bytes input bytes apart things get ugly!

des_read_pw_string() is the most machine/OS dependent function and normally generates the most problems when porting this code.

CONFORMING TO

ANSI X3.106

SEE ALSO

crypt (3), des_modes(7), evp(3), rand(3)

HISTORY

des_cbc_cksum(), *des_cbc_encrypt()*, *des_ecb_encrypt()*, *des_is_weak_key()*, *des_key_sched()*, *des_pcbc_encrypt()*, *des_quad_cksum()*, *des_random_key()*, *des_read_password()* and *des_check_key_parity()*, *des_fixup_key_parity()* and *des_is_weak_key()* are available in newer versions of that library.

des_set_key_checked() and *des_set_key_unchecked()* were added in OpenSSL 0.9.5.

des_generate_random_block(), *des_init_random_number_generator()*, *des_new_random_key()*, *des_set_random_generator_seed()* and *des_set_sequence_number()* and *des_rand_data()* are used in newer versions of Kerberos but are not implemented here.

des_random_key() generated cryptographically weak random data in SSLeay and in OpenSSL prior version 0.9.5, as well as in the original MIT library.

AUTHOR

Eric Young (eay@cryptsoft.com). Modified for the OpenSSL project (http://www.openssl.org).

NAME

devname — get device name

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>
#include <sys/stat.h>

char *
devname(dev_t dev, mode_t type);
```

DESCRIPTION

The **devname**() function returns a pointer to the name of the block or character device in "/dev" with a device number of *dev*, and a file type matching the one encoded in *type* which must be one of S_IFBLK or S_IFCHR. The device name is cached so that multiple calls with the same *dev* and *type* do not require additional queries of the device database file. If no device matches the specified values, or no information is available, NULL is returned.

The traditional display for applications when no device is found is the string "??".

FILES

/var/run/dev.db Device database file.

SEE ALSO

stat(2), dev_mkdb(8)

HISTORY

The **devname** function call appeared in 4.4BSD.

NAME

dhcpctl_initialize — dhcpctl library initialization

SYNOPSIS

```
#include <dhcpctl/dhcpctl.h>

dhcpctl_status
dhcpctl_initialize(void);

dhcpctl_status
dhcpctl_connect(dhcpctl_handle *cxn, const char *host, int port,
    dhcpctl_handle auth);

dhcpctl_status
dhcpctl_wait_for_completion(dhcpctl_handle object, dhcpctl_status *status);

dhcpctl_status
dhcpctl_get_value(dhcpctl_data_string *value, dhcpctl_handle object,
    const char *name);

dhcpctl_status
dhcpctl_get_boolean(int *value, dhcpctl_handle object, const char *name);

dhcpctl_status
dhcpctl_set_value(dhcpctl_handle object, dhcpctl_data_string value,
    const char *name);

dhcpctl_status
dhcpctl_set_string_value(dhcpctl_handle object, const char *value,
    const char *name);

dhcpctl_status
dhcpctl_set_boolean_value(dhcpctl_handle object, int value,
    const char *name);

dhcpctl_status
dhcpctl_set_int_value(dhcpctl_handle object, int value, const char *name);

dhcpctl_status
dhcpctl_object_update(dhcpctl_handle connection, dhcpctl_handle object);

dhcpctl_status
dhcpctl_object_refresh(dhcpctl_handle connection, dhcpctl_handle object);

dhcpctl_status
dhcpctl_object_remove(dhcpctl_handle connection, dhcpctl_handle object);

dhcpctl_status
dhcpctl_set_callback(dhcpctl_handle object, void *data,
    void (*function) (dhcpctl_handle, dhcpctl_status, void *));

dhcpctl_status
dhcpctl_new_authenticator(dhcpctl_handle *object, const char *name,
    const char *algorithm, const char *secret, unsigned secret_len);

dhcpctl_status
dhcpctl_new_object(dhcpctl_handle *object, dhcpctl_handle connection,
    const char *object_type);
```

dhcpctl_status
dhcpctl_open_object(*dhcpctl_handle object*, *dhcpctl_handle connection*, *int flags*);

isc_result_t
omapi_data_string_new(*dhcpctl_data_string*, **data*, *unsigned*, *int*, *length*, *const*, *char*, **filename,*, *int*, *lineno*);

isc_result_t
dhcpctl_data_string_dereference(*dhcpctl_data_string **, *const char **, *int*);

DESCRIPTION

The dhcpctl set of functions provide an API that can be used to communicate with and manipulate a running ISC DHCP server. All functions return a value of `isc_result_t`. The return values reflects the result of operations to local data structures. If an operation fails on the server for any reason, then the error result will be returned through the second parameter of the **dhcpctl_wait_for_completion**() call.

dhcpctl_initialize() sets up the data structures the library needs to do its work. This function must be called once before any other.

dhcpctl_connect() opens a connection to the DHCP server at the given host and port. If an authenticator has been created for the connection, then it is given as the 4th argument. On a successful return the address pointed at by the first argument will have a new connection object assigned to it.

For example:

```
s = dhcpctl_connect(&cxn, "127.0.0.1", 7911, NULL);
```

connects to the DHCP server on the localhost via port 7911 (the standard OMAPI port). No authentication is used for the connection.

dhcpctl_wait_for_completion() flushes a pending message to the server and waits for the response. The result of the request as processed on the server is returned via the second parameter.

```
s = dhcpctl_wait_for_completion(cxn, &wv);
if (s != ISC_R_SUCCESS)
        local_failure(s);
else if (wv != ISC_R_SUCCESS)
        server_failure(wc);
```

The call to **dhcpctl_wait_for_completion**() won't return until the remote message processing completes or the connection to the server is lost.

dhcpctl_get_value() extracts a value of an attribute from the handle. The value can be of any length and is treated as a sequence of bytes. The handle must have been created first with **dhcpctl_new_object**() and opened with **dhcpctl_open_object**(). The value is returned via the parameter named "value". The last parameter is the name of attribute to retrieve.

```
dhcpctl_data_string value = NULL;
dhcpctl_handle lease;
time_t thetime;

s = dhcpctl_get_value (&value, lease, "ends");
assert(s == ISC_R_SUCCESS && value->len == sizeof(thetime));
memcpy(&thetime, value->value, value->len);
```

dhcpctl_get_boolean() extracts a boolean valued attribute from the object handle.

The **dhcpctl_set_value**(), **dhcpctl_set_string_value**(), **dhcpctl_set_boolean_value**(), and **dhcpctl_set_int_value**() functions all set a value on the object handle.

dhcpctl_object_update() function queues a request for all the changes made to the object handle be be sent to the remote for processing. The changes made to the atributes on the handle will be applied to remote object if permitted.

dhcpctl_object_refresh() queues up a request for a fresh copy of all the attribute values to be sent from the remote to refresh the values in the local object handle.

dhcpctl_object_remove() queues a request for the removal on the server of the object referenced by the handle.

The **dhcpctl_set_callback**() function sets up a user-defined function to be called when an event completes on the given object handle. This is needed for asynchronous handling of events, versus the synchronous handling given by **dhcpctl_wait_for_completion**(). When the function is called the first parameter is the object the event arrived for, the second is the status of the message that was processed, the third is the same value as the second parameter given to **dhcpctl_set_callback**().

The **dhcpctl_new_authenticator**() creates a new authenticator object to be used for signing the messages that cross over the network. The “name”, “algorithm”, and “secret” values must all match what the server uses and are defined in its configuration file. The created object is returned through the first parameter and must be used as the 4th parameter to **dhcpctl_connect**(). Note that the ’secret’ value must not be base64 encoded, which is different from how the value appears in the dhcpd.conf file.

dhcpctl_new_object() creates a local handle for an object on the the server. The “object_type” parameter is the ascii name of the type of object being accessed. e.g. "lease". This function only sets up local data structures, it does not queue any messages to be sent to the remote side, **dhcpctl_open_object**() does that.

dhcpctl_open_object() builds and queues the request to the remote side. This function is used with handle created via **dhcpctl_new_object**(). The flags argument is a bit mask with the following values available for setting:

DHCPCTL_CREATE
: if the object does not exist then the remote will create it

DHCPCTL_UPDATE
: update the object on the remote side using the attributes already set in the handle.

DHCPCTL_EXCL
: return and error if the object exists and DHCPCTL_CREATE was also specified

The **omapi_data_string_new**() function allocates a new *dhcpctl_data_string* object. The data string will be large enough to hold “length” bytes of data. The “file” and “lineno” arguments are the source file location the call is made from, typically by using the __FILE__ and __LINE__ macros or the MDL macro defined in

dhcpctl_data_string_dereference() deallocates a data string created by **omapi_data_string_new**(). The memory for the object won’t be freed until the last reference is released.

EXAMPLES

The following program will connect to the DHCP server running on the local host and will get the details of the existing lease for IP address 10.0.0.101. It will then print out the time the lease is due to expire. Note that most error checking has been ommitted for brevity.

```
#include <stdarg.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>

#include <isc/result.h>
#include <dhcpctl/dhcpctl.h>

int main (int argc, char **argv) {
        dhcpctl_data_string ipaddrstring = NULL;
        dhcpctl_data_string value = NULL;
        dhcpctl_handle connection = NULL;
        dhcpctl_handle lease = NULL;
        isc_result_t waitstatus;
        struct in_addr convaddr;
        time_t thetime;

        dhcpctl_initialize ();

        dhcpctl_connect (&connection, "127.0.0.1",
                         7911, 0);

        dhcpctl_new_object (&lease, connection,
                            "lease");

        memset (&ipaddrstring, 0, sizeof
                ipaddrstring);

        inet_pton(AF_INET, "10.0.0.101",
                  &convaddr);

        omapi_data_string_new (&ipaddrstring,
                               4, MDL);
        memcpy(ipaddrstring->value, &convaddr.s_addr, 4);

        dhcpctl_set_value (lease, ipaddrstring,
                           "ip-address");

        dhcpctl_open_object (lease, connection, 0);

        dhcpctl_wait_for_completion (lease,
                                     &waitstatus);
        if (waitstatus != ISC_R_SUCCESS) {
                /* server not authoritative */
                exit (0);
        }

        dhcpctl_data_string_dereference(&ipaddrstring,
                                        MDL);

        dhcpctl_get_value (&value, lease, "ends");
```

```
                memcpy(&thetime, value->value, value->len);

                dhcpctl_data_string_dereference(&value, MDL);

                fprintf (stdout, "ending time is %s",
                         ctime(&thetime));
        }
```

SEE ALSO

omshell(1), dhclient.conf(5), dhcpd.conf(5), dhclient(8), dhcpd(8)

AUTHORS

dhcpctl_initialize was written by Ted Lemon of Nominum, Inc. This preliminary documentation was written by James Brister of Nominum, Inc.

NAME

opendir, **readdir**, **readdir_r**, **telldir**, **seekdir**, **rewinddir**, **closedir**, **dirfd** — directory operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <dirent.h>

DIR *
opendir(const char *filename);

struct dirent *
readdir(DIR *dirp);

int
readdir_r(DIR * restrict dirp, struct dirent * restrict entry,
    struct dirent ** restrict result);

long
telldir(DIR *dirp);

void
seekdir(DIR *dirp, long loc);

void
rewinddir(DIR *dirp);

int
closedir(DIR *dirp);

int
dirfd(DIR *dirp);
```

DESCRIPTION

The **opendir**() function opens the directory named by *filename*, associates a *directory stream* with it and returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer NULL is returned if *filename* cannot be accessed, or if it cannot malloc(3) enough memory to hold the whole thing.

The **readdir**() function returns a pointer to the next directory entry. It returns NULL upon reaching the end of the directory or detecting an invalid **seekdir**() operation.

The **readdir_r**() function provides the same functionality as **readdir**(), but the caller must provide a directory *entry* buffer to store the results in. If the read succeeds, *result* is pointed at the *entry*; upon reaching the end of the directory *result* is set to NULL. The **readdir_r**() function returns 0 on success or an error number to indicate failure.

The **telldir**() function returns the current location associated with the named *directory stream*.

The **seekdir**() function sets the position of the next **readdir**() operation on the *directory stream*. The new position reverts to the one associated with the *directory stream* when the **telldir**() operation was performed. Values returned by **telldir**() are good only for the lifetime of the DIR pointer, *dirp*, from which they are derived. If the directory is closed and then reopened, the **telldir**() value cannot be reused.

The **rewinddir**() function resets the position of the named *directory stream* to the beginning of the directory.

The **closedir**() function closes the named *directory stream* and frees the structure associated with the *dirp* pointer, returning 0 on success. On failure, −1 is returned and the global variable *errno* is set to indicate the error.

The **dirfd**() function returns the integer file descriptor associated with the named *directory stream*, see open(2).

EXAMPLES

Sample code which searches a directory for entry “name” is:

```
len = strlen(name);
dirp = opendir(".");
if (dirp != NULL) {
        while ((dp = readdir(dirp)) != NULL)
                if (dp->d_namlen == len &&
                    !strcmp(dp->d_name, name)) {
                        (void)closedir(dirp);
                        return (FOUND);
                }
        (void)closedir(dirp);
}
return (NOT_FOUND);
```

SEE ALSO

close(2), lseek(2), open(2), read(2), dir(5)

STANDARDS

The **opendir**(), **readdir**(), **rewinddir**() and **closedir**() functions conform to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

The **opendir**(), **readdir**(), **telldir**(), **seekdir**(), **rewinddir**(), **closedir**(), and **dirfd**() functions appeared in 4.2BSD.

NAME

dirname — report the parent directory name of a file pathname

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <libgen.h>

char *
dirname(char *path);
```

DESCRIPTION

The **dirname**() function takes a pointer to a character string that contains a pathname, *path*, and returns a pointer to a string that is a pathname of the parent directory of *path*. Trailing '/' characters in *path* are not counted as part of the path.

If *path* does not contain a '/', then **dirname**() returns a pointer to the string ".".

If *path* is a null pointer or points to an empty string, **dirname**() returns a pointer to the string ".".

RETURN VALUES

The **dirname**() function returns a pointer to a string that is the parent directory of *path*.

SEE ALSO

dirname(1), basename(3)

STANDARDS

- X/Open Portability Guide Issue 4, Version 2 ("XPG4.2")
- IEEE Std 1003.1-2001 ("POSIX.1")

BUGS

If the length of the result is longer than PATH_MAX bytes (including the terminating nul), the result will be truncated.

The **dirname**() function returns a pointer to static storage that may be overwritten by subsequent calls to **dirname**(). This is not strictly a bug; it is explicitly allowed by IEEE Std 1003.1-2001 ("POSIX.1").

NAME

disklabel_dkcksum — compute the checksum for a disklabel

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

uint16_t
disklabel_dkcksum(struct disklabel *lp);
```

DESCRIPTION

disklabel_dkcksum() computes the checksum for the disklabel passed in as *lp*.

RETURN VALUES

The **disklabel_dkcksum**() returns the computed checksum.

HISTORY

The **disklabel_dkcksum** function call appeared in NetBSD 2.0.

NAME

disklabel_scan — scan a buffer for a valid disklabel

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
disklabel_scan(struct disklabel *lp, char *buf, size_t buflen);
```

DESCRIPTION

disklabel_scan() scans the memory region specified by *buf* and *buflen* for a valid disklabel. If such a label is found, it is copied into *lp*.

RETURN VALUES

The **disklabel_scan**() function returns 0 if a valid disklabel was found and 1 if not.

HISTORY

The **disklabel_scan** function call appeared in NetBSD 2.0.

NAME

div — return quotient and remainder from division

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

div_t
div(int num, int denom);
```

DESCRIPTION

The **div**() function computes the value *num/denom* and returns the quotient and remainder in a structure named *div_t* that contains two *int* members named *quot* and *rem*.

SEE ALSO

ldiv(3), lldiv(3), math(3), qdiv(3)

STANDARDS

The **div**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

dlopen, **dlclose**, **dlsym**, **dladdr**, **dlctl**, **dlerror** — dynamic link interface

LIBRARY

(These functions are not in a library. They are included in every dynamically linked program automatically.)

SYNOPSIS

```
#include <dlfcn.h>

void *
dlopen(const char *path, int mode);

int
dlclose(void *handle);

void *
dlsym(void * restrict handle, const char * restrict symbol);

int
dladdr(void * restrict addr, Dl_info * restrict dli);

int
dlctl(void *handle, int cmd, void *data);

char *
dlerror(void);
```

DESCRIPTION

These functions provide an interface to the run-time linker `ld.so`(1). They allow new shared objects to be loaded into the process' address space under program control. The **dlopen**() function takes a name of a shared object as the first argument. The shared object is mapped into the address space, relocated and its external references are resolved in the same way as is done with the implicitly loaded shared libraries at program startup. The argument can either be an absolute pathname or it can be of the form "lib⟨name⟩.so[.xx[.yy]]" in which case the same library search rules apply that are used for "intrinsic" shared library searches. If the first argument is `NULL`, **dlopen**() returns a handle on the global symbol object. This object provides access to all symbols from an ordered set of objects consisting of the original program image and any dependencies loaded during startup.

The second argument has currently no effect, but should be set to `RTLD_LAZY` for future compatibility. **dlopen**() returns a handle to be used in calls to **dlclose**(), **dlsym**() and **dlctl**(). If the named shared object has already been loaded by a previous call to **dlopen**() (and not yet unloaded by **dlclose**()), a handle referring to the resident copy is returned.

dlclose() unlinks and removes the object referred to by *handle* from the process address space. If multiple calls to **dlopen**() have been done on this object (or the object was one loaded at startup time) the object is removed when its reference count drops to zero.

dlsym() looks for a definition of *symbol* in the shared object designated by *handle*. The symbols address is returned. If the symbol cannot be resolved, `NULL` is returned.

dladdr() examines all currently mapped shared objects for a symbol whose address -- as mapped in the process address space -- is closest to but not exceeding the value passed in the first argument *addr*. The symbols of a shared object are only eligible if *addr* is between the base address of the shared object and the value of the symbol "_end" in the same shared object. If no object for which this condition holds true can be found, **dladdr**() will return 0. Otherwise, a non-zero value is returned and the *dli* argument will be used to provide information on the selected symbol and the shared object it is contained in. The *dli* argument points at a caller-provided *Dl_info* structure defined as follows:

```
typedef struct {
        const char  *dli_fname;     /* File defining the symbol */
        void        *dli_fbase;     /* Base address */
        const char  *dli_sname;     /* Symbol name */
        const void  *dli_saddr;     /* Symbol address */
} Dl_info;
```

The member *dli_sname* points at the nul-terminated name of the selected symbol, and *dli_saddr* is the actual address (as it appears in the process address space) of the symbol. The member *dli_fname* points at the file name corresponding to the shared object in which the symbol was found, while *dli_fbase* is the base address at which this shared object is loaded in the process address space. *dli_fname* and *dli_fbase* may be zero if the symbol was found in the internally generated "copy" section (see `link`(5)) which is not associated with a file. Note: both strings pointed at by *dli_fname* and *dli_sname* reside in memory private to the run-time linker module and should not be modified by the caller.

`dlctl`() provides an interface similar to `ioctl`(2) to control several aspects of the run-time linker's operation. This interface is currently under development.

`dlerror`() returns a character string representing the most recent error that has occurred while processing one of the other functions described here. If no dynamic linking errors have occurred since the last invocation of **`dlerror`**(), **`dlerror`**() returns `NULL`. Thus, invoking **`dlerror`**() a second time, immediately following a prior invocation, will result in `NULL` being returned.

SEE ALSO

`ld`(1), `rtld`(1), `link`(5)

HISTORY

Some of the **`dl`**∗ functions first appeared in SunOS 4.

BUGS

An error that occurs while processing a **`dlopen`**() request results in the termination of the program.

NAME

ecalloc, **emalloc**, **eread**, **erealloc**, **esetenv**, **estrdup**, **ewrite** — exit-on-failure wrapper functions

LIBRARY

The roken library (libroken, -lroken)

SYNOPSIS

```
#include <roken.h>

void *
ecalloc(size_t number, size_t size);

void *
emalloc(size_t sz);

ssize_t
eread(int fd, void *buf, size_t nbytes);

void *
erealloc(void *ptr, size_t sz);

void
esetenv(const char *var, const char *val, int rewrite);

char *
estrdup(const char *str);

ssize_t
ewrite(int fd, const void *buf, size_t nbytes);
```

DESCRIPTION

These functions do the same as the ones without the “e” prefix, but if there is an error they will print a message with errx(3), and exit. For **eread** and **ewrite** this is also true for partial data.

This is useful in applications when there is no need for a more advanced failure mode.

SEE ALSO

read(2), write(2), calloc(3), errx(3), malloc(3), realloc(3), setenv(3), strdup(3)

NAME

editline − command-line editing library with history

SYNOPSIS

```
char *
readline(prompt)
    char  *prompt;

void
add_history(line)
   char   *line;
```

DESCRIPTION

Editline is a library that provides an line-editing interface with text recall. It is intended to be compatible with the *readline* library provided by the Free Software Foundation, but much smaller. The bulk of this manual page describes the user interface.

The *readline* routine returns a line of text with the trailing newline removed. The data is returned in a buffer allocated with *malloc*(3), so the space should be released with *free*(3) when the calling program is done with it. Before accepting input from the user, the specified *prompt* is displayed on the terminal.

The *add_history* routine makes a copy of the specified *line* and adds it to the internal history list.

User Interface

A program that uses this library provides a simple emacs-like editing interface to its users. A line may be edited before it is sent to the calling program by typing either control characters or escape sequences. A control character, shown as a caret followed by a letter, is typed by holding down the ‘‘control’’ key while the letter is typed. For example, ‘‘^A’’ is a control-A. An escape sequence is entered by typing the ‘‘escape’’ key followed by one or more characters. The escape key is abbreviated as ‘‘ESC.’’ Note that unlike control keys, case matters in escape sequences; ‘‘ESC F’’ is not the same as ‘‘ESC f’’.

An editing command may be typed anywhere on the line, not just at the beginning. In addition, a return may also be typed anywhere on the line, not just at the end.

Most editing commands may be given a repeat count, *n*, where *n* is a number. To enter a repeat count, type the escape key, the number, and then the command to execute. For example, ‘‘ESC 4 ^f’’ moves forward four characters. If a command may be given a repeat count then the text ‘‘[n]’’ is given at the end of its description.

The following control characters are accepted:

^A	Move to the beginning of the line
^B	Move left (backwards) [n]
^D	Delete character [n]
^E	Move to end of line
^F	Move right (forwards) [n]
^G	Ring the bell
^H	Delete character before cursor (backspace key) [n]
^I	Complete filename (tab key); see below
^J	Done with line (return key)
^K	Kill to end of line (or column [n])
^L	Redisplay line
^M	Done with line (alternate return key)
^N	Get next line from history [n]
^P	Get previous line from history [n]
^R	Search backward (forward if [n]) through history for text; must start line if text begins with an uparrow
^T	Transpose characters
^V	Insert next character, even if it is an edit command
^W	Wipe to the mark

^X^X	Exchange current location and mark
^Y	Yank back last killed text
^[	Start an escape sequence (escape key)
^]c	Move forward to next character ‘‘c’’
^?	Delete character before cursor (delete key) [n]

The following escape sequences are provided.

ESC ^H	Delete previous word (backspace key) [n]
ESC DEL	Delete previous word (delete key) [n]
ESC SP	Set the mark (space key); see ^X^X and ^Y above
ESC .	Get the last (or [n]’th) word from previous line
ESC ?	Show possible completions; see below
ESC <	Move to start of history
ESC >	Move to end of history
ESC b	Move backward a word [n]
ESC d	Delete word under cursor [n]
ESC f	Move forward a word [n]
ESC l	Make word lowercase [n]
ESC u	Make word uppercase [n]
ESC y	Yank back last killed text
ESC v	Show library version
ESC w	Make area up to mark yankable
ESC nn	Set repeat count to the number nn
ESC C	Read from environment variable ‘‘_C_’’, where C is an uppercase letter

The *editline* library has a small macro facility. If you type the escape key followed by an uppercase letter, *C*, then the contents of the environment variable *_C_* are read in as if you had typed them at the keyboard. For example, if the variable *_L_* contains the following:

^A^Kecho ’^V^[[H^V^[[2J’^M

Then typing ‘‘ESC L’’ will move to the beginning of the line, kill the entire line, enter the echo command needed to clear the terminal (if your terminal is like a VT-100), and send the line back to the shell.

The *editline* library also does filename completion. Suppose the root directory has the following files in it:

bin vmunix
core vmunix.old

If you type ‘‘rm /v’’ and then the tab key. *Editline* will then finish off as much of the name as possible by adding ‘‘munix’’. Because the name is not unique, it will then beep. If you type the escape key and a question mark, it will display the two choices. If you then type a period and a tab, the library will finish off the filename for you:

rm /v[TAB]*munix*.TAB*old*

The tab key is shown by ‘‘[TAB]’’ and the automatically-entered text is shown in italics.

BUGS AND LIMITATIONS

Cannot handle lines more than 80 columns.

AUTHORS

Simmule R. Turner <uunet.uu.net!capitol!sysgo!simmy> and Rich $alz <rsalz@osf.org>. Original manual page by DaviD W. Sanderson <dws@ssec.wisc.edu>.

NAME

editline, **el_init**, **el_end**, **el_reset**, **el_gets**, **el_getc**, **el_push**, **el_parse**, **el_set**, **el_get**, **el_source**, **el_resize**, **el_line**, **el_insertstr**, **el_deletestr**, **history_init**, **history_end**, **history**, **tok_init**, **tok_end**, **tok_reset**, **tok_line**, **tok_str** — line editor, history and tokenization functions

LIBRARY

Command Line Editor Library (libedit, −ledit)

SYNOPSIS

```
#include <histedit.h>

EditLine *
el_init(const char *prog, FILE *fin, FILE *fout, FILE *ferr);

void
el_end(EditLine *e);

void
el_reset(EditLine *e);

const char *
el_gets(EditLine *e, int *count);

int
el_getc(EditLine *e, char *ch);

void
el_push(EditLine *e, const char *str);

int
el_parse(EditLine *e, int argc, const char *argv[]);

int
el_set(EditLine *e, int op, ...);

int
el_get(EditLine *e, int op, ...);

int
el_source(EditLine *e, const char *file);

void
el_resize(EditLine *e);

const LineInfo *
el_line(EditLine *e);

int
el_insertstr(EditLine *e, const char *str);

void
el_deletestr(EditLine *e, int count);

History *
history_init();

void
history_end(History *h);
```

int
history(*History* ∗*h*, *HistEvent* ∗*ev*, *int op*, *...*);

Tokenizer ∗
tok_init(*const char* ∗*IFS*);

void
tok_end(*Tokenizer* ∗*t*);

void
tok_reset(*Tokenizer* ∗*t*);

int
tok_line(*Tokenizer* ∗*t*, *const LineInfo* ∗*li*, *int* ∗*argc*, *const char* ∗∗*argv[]*, *int* ∗*cursorc*, *int* ∗*cursoro*);

int
tok_str(*Tokenizer* ∗*t*, *const char* ∗*str*, *int* ∗*argc*, *const char* ∗∗*argv[]*);

DESCRIPTION

The **editline** library provides generic line editing, history and tokenization functions, similar to those found in sh(1).

These functions are available in the **libedit** library (which needs the **libtermcap** library). Programs should be linked with **−ledit −ltermcap**.

LINE EDITING FUNCTIONS

The line editing functions use a common data structure, *EditLine*, which is created by **el_init**() and freed by **el_end**().

The following functions are available:

el_init()
Initialise the line editor, and return a data structure to be used by all other line editing functions. *prog* is the name of the invoking program, used when reading the editrc(5) file to determine which settings to use. *fin*, *fout* and *ferr* are the input, output, and error streams (respectively) to use. In this documentation, references to "the tty" are actually to this input/output stream combination.

el_end()
Clean up and finish with *e*, assumed to have been created with **el_init**().

el_reset()
Reset the tty and the parser. This should be called after an error which may have upset the tty's state.

el_gets()
Read a line from the tty. *count* is modified to contain the number of characters read. Returns the line read if successful, or NULL if no characters were read or if an error occurred.

el_getc()
Read a character from the tty. *ch* is modified to contain the character read. Returns the number of characters read if successful, −1 otherwise.

el_push()
Pushes *str* back onto the input stream. This is used by the macro expansion mechanism. Refer to the description of **bind −s** in editrc(5) for more information.

el_parse()
Parses the *argv* array (which is *argc* elements in size) to execute builtin **editline** commands. If the command is prefixed with "prog": then **el_parse**() will only execute the command if "prog"

matches the *prog* argument supplied to **el_init**(). The return value is −1 if the command is unknown, 0 if there was no error or "prog" didn't match, or 1 if the command returned an error. Refer to editrc(5) for more information.

el_set()
Set **editline** parameters. *op* determines which parameter to set, and each operation has its own parameter list.

The following values for *op* are supported, along with the required argument list:

EL_PROMPT, *char *(*f)(EditLine *)*
Define prompt printing function as *f*, which is to return a string that contains the prompt.

EL_REFRESH
Re-display the current line on the next terminal line.

EL_RPROMPT, *char *(*f)(EditLine *)*
Define right side prompt printing function as *f*, which is to return a string that contains the prompt.

EL_TERMINAL, *const char *type*
Define terminal type of the tty to be *type*, or to TERM if *type* is NULL.

EL_EDITOR, *const char *mode*
Set editing mode to *mode*, which must be one of "emacs" or "vi".

EL_SIGNAL, *int flag*
If *flag* is non-zero, **editline** will install its own signal handler for the following signals when reading command input: SIGCONT, SIGHUP, SIGINT, SIGQUIT, SIGSTOP, SIGTERM, SIGTSTP, and SIGWINCH. Otherwise, the current signal handlers will be used.

EL_BIND, *const char **, *. . .*, NULL
Perform the **bind** builtin command. Refer to editrc(5) for more information.

EL_ECHOTC, *const char **, *. . .*, NULL
Perform the **echotc** builtin command. Refer to editrc(5) for more information.

EL_SETTC, *const char **, *. . .*, NULL
Perform the **settc** builtin command. Refer to editrc(5) for more information.

EL_SETTY, *const char **, *. . .*, NULL
Perform the **setty** builtin command. Refer to editrc(5) for more information.

EL_TELLTC, *const char **, *. . .*, NULL
Perform the **telltc** builtin command. Refer to editrc(5) for more information.

EL_ADDFN, *const char *name*, *const char *help*, *unsigned char (*func)(EditLine *e, int ch)*
Add a user defined function, **func**(), referred to as *name* which is invoked when a key which is bound to *name* is entered. *help* is a description of *name*. At invocation time, *ch* is the key which caused the invocation. The return value of **func**() should be one of:

CC_NORM	Add a normal character.
CC_NEWLINE	End of line was entered.
CC_EOF	EOF was entered.
CC_ARGHACK	Expecting further command input as arguments, do nothing visually.

CC_REFRESH Refresh display.

CC_REFRESH_BEEP
Refresh display, and beep.

CC_CURSOR Cursor moved, so update and perform CC_REFRESH.

CC_REDISPLAY Redisplay entire input line. This is useful if a key binding outputs extra information.

CC_ERROR An error occurred. Beep, and flush tty.

CC_FATAL Fatal error, reset tty to known state.

EL_HIST, *History *(*func)(History *, int op, ...)*, *const char *ptr*
Defines which history function to use, which is usually **history**(). *ptr* should be the value returned by **history_init**().

EL_EDITMODE, *int flag*
If *flag* is non-zero, editing is enabled (the default). Note that this is only an indication, and does not affect the operation of **editline**. At this time, it is the caller's responsibility to check this (using **el_get**()) to determine if editing should be enabled or not.

EL_GETCFN, *int (*f)(EditLine *, char *c)*
Define the character reading function as *f*, which is to return the number of characters read and store them in *c*. This function is called internally by **el_gets**() and **el_getc**(). The builtin function can be set or restored with the special function name ''EL_BUILTIN_GETCFN''.

EL_CLIENTDATA, *void *data*
Register *data* to be associated with this EditLine structure. It can be retrieved with the corresponding **el_get**() call.

EL_SETFP, *int fd*, *FILE *fp*
Set the current **editline** file pointer for "input" *fd* = 0, "output" *fd* = 1, or "error" *fd* = 2 from *fp*.

el_get()
Get **editline** parameters. *op* determines which parameter to retrieve into *result*. Returns 0 if successful, −1 otherwise.

The following values for *op* are supported, along with actual type of *result*:

EL_PROMPT, *char *(*f)(EditLine *)*
Return a pointer to the function that displays the prompt.

EL_RPROMPT, *char *(*f)(EditLine *)*
Return a pointer to the function that displays the rightside prompt.

EL_EDITOR, *const char **
Return the name of the editor, which will be one of "emacs" or "vi".

EL_GETTC, *const char *name*, *void *value*
Return non-zero if *name* is a valid termcap(5) capability and set *value* to the current value of that capability.

EL_SIGNAL, *int **
Return non-zero if **editline** has installed private signal handlers (see **el_get**() above).

EL_EDITMODE, *int **
Return non-zero if editing is enabled.

EL_GETCFN, *int (**f)(EditLine *, char *)*
Return a pointer to the function that read characters, which is equal to ‘‘EL_BUILTIN_GETCFN’’ in the case of the default builtin function.

EL_CLIENTDATA, *void **data*
Retrieve *data* previously registered with the corresponding **el_set**() call.

EL_UNBUFFERED, *int*
Sets or clears unbuffered mode. In this mode, **el_gets**() will return immediately after processing a single character.

EL_PREP_TERM, *int*
Sets or clears terminal editing mode.

EL_GETFP, *int fd*, *FILE **fp*
Return in *fp* the current **editline** file pointer for “input” *fd* = 0, “output” *fd* = 1, or “error” *fd* = 2.

el_source()
Initialise **editline** by reading the contents of *file*. **el_parse**() is called for each line in *file*. If *file* is NULL, try $PWD/.editrc then $HOME/.editrc. Refer to editrc(5) for details on the format of *file*.

el_resize()
Must be called if the terminal size changes. If EL_SIGNAL has been set with **el_set**(), then this is done automatically. Otherwise, it’s the responsibility of the application to call **el_resize**() on the appropriate occasions.

el_line()
Return the editing information for the current line in a *LineInfo* structure, which is defined as follows:

```
typedef struct lineinfo {
    const char *buffer;    /* address of buffer */
    const char *cursor;    /* address of cursor */
    const char *lastchar;  /* address of last character */
} LineInfo;
```

buffer is not NUL terminated. This function may be called after **el_gets**() to obtain the *LineInfo* structure pertaining to line returned by that function, and from within user defined functions added with EL_ADDFN.

el_insertstr()
Insert *str* into the line at the cursor. Returns −1 if *str* is empty or won’t fit, and 0 otherwise.

el_deletestr()
Delete *count* characters before the cursor.

HISTORY LIST FUNCTIONS

The history functions use a common data structure, *History*, which is created by **history_init**() and freed by **history_end**().

The following functions are available:

history_init()
Initialise the history list, and return a data structure to be used by all other history list functions.

history_end()
Clean up and finish with *h*, assumed to have been created with **history_init**().

history()
Perform operation *op* on the history list, with optional arguments as needed by the operation. *ev* is changed accordingly to operation. The following values for *op* are supported, along with the required argument list:

H_SETSIZE, *int size*
Set size of history to *size* elements.

H_GETSIZE
Get number of events currently in history.

H_END
Cleans up and finishes with *h*, assumed to be created with **history_init**().

H_CLEAR
Clear the history.

H_FUNC, *void *ptr*, *history_gfun_t first*, *history_gfun_t next*, *history_gfun_t last*, *history_gfun_t prev*, *history_gfun_t curr*, *history_sfun_t set*, *history_vfun_t clear*, *history_efun_t enter*, *history_efun_t add*
Define functions to perform various history operations. *ptr* is the argument given to a function when it's invoked.

H_FIRST
Return the first element in the history.

H_LAST
Return the last element in the history.

H_PREV
Return the previous element in the history.

H_NEXT
Return the next element in the history.

H_CURR
Return the current element in the history.

H_SET
Set the cursor to point to the requested element.

H_ADD, *const char *str*
Append *str* to the current element of the history, or perform the H_ENTER operation with argument *str* if there is no current element.

H_APPEND, *const char *str*
Append *str* to the last new element of the history.

H_ENTER, *const char *str*
Add *str* as a new element to the history, and, if necessary, removing the oldest entry to keep the list to the created size. If H_SETUNIQUE was has been called with a non-zero arguments, the element will not be entered into the history if its contents match the ones of the current history element. If the element is entered **history**() returns 1, if it is ignored as a duplicate returns 0. Finally **history**() returns −1 if an error occurred.

H_PREV_STR, *const char *str*
Return the closest previous event that starts with *str*.

H_NEXT_STR, *const char *str*
Return the closest next event that starts with *str*.

H_PREV_EVENT, *int e*
Return the previous event numbered *e*.

H_NEXT_EVENT, *int e*
Return the next event numbered *e*.

H_LOAD, *const char *file*
Load the history list stored in *file*.

H_SAVE, *const char *file*
Save the history list to *file*.

H_SETUNIQUE, *int unique*
Set flag that adjacent identical event strings should not be entered into the history.

H_GETUNIQUE
Retrieve the current setting if adjacent identical elements should be entered into the history.

H_DEL, *int e*
Delete the event numbered *e*. This function is only provided for readline(3) compatibility. The caller is responsible for free'ing the string in the returned *HistEvent*.

history() returns >= 0 if the operation *op* succeeds. Otherwise, −1 is returned and *ev* is updated to contain more details about the error.

TOKENIZATION FUNCTIONS

The tokenization functions use a common data structure, *Tokenizer*, which is created by **tok_init**() and freed by **tok_end**().

The following functions are available:

tok_init()
Initialise the tokenizer, and return a data structure to be used by all other tokenizer functions. *IFS* contains the Input Field Separators, which defaults to ⟨space⟩, ⟨tab⟩, and ⟨newline⟩ if NULL.

tok_end()
Clean up and finish with *t*, assumed to have been created with **tok_init**().

tok_reset()
Reset the tokenizer state. Use after a line has been successfully tokenized by **tok_line**() or **tok_str**() and before a new line is to be tokenized.

tok_line()
Tokenize *li*, If successful, modify: *argv* to contain the words, *argc* to contain the number of words, *cursorc* (if not NULL) to contain the index of the word containing the cursor, and *cursoro* (if not NULL) to contain the offset within *argv[cursorc]* of the cursor.

Returns 0 if successful, −1 for an internal error, 1 for an unmatched single quote, 2 for an unmatched double quote, and 3 for a backslash quoted ⟨newline⟩. A positive exit code indicates that another line should be read and tokenization attempted again.

tok_str()
A simpler form of **tok_line**(); *str* is a NUL terminated string to tokenize.

SEE ALSO

sh(1), signal(3), termcap(3), editrc(5), termcap(5)

HISTORY

The **editline** library first appeared in 4.4BSD. CC_REDISPLAY appeared in NetBSD 1.3. CC_REFRESH_BEEP, EL_EDITMODE and the readline emulation appeared in NetBSD 1.4. EL_RPROMPT appeared in NetBSD 1.5.

AUTHORS

The **editline** library was written by Christos Zoulas. Luke Mewburn wrote this manual and implemented CC_REDISPLAY, CC_REFRESH_BEEP, EL_EDITMODE, and EL_RPROMPT. Jaromir Dolecek implemented the readline emulation.

BUGS

At this time, it is the responsibility of the caller to check the result of the EL_EDITMODE operation of **el_get**() (after an **el_source**() or **el_parse**()) to determine if **editline** should be used for further input. I.e., EL_EDITMODE is purely an indication of the result of the most recent editrc(5) **edit** command.

NAME

esetfunc, **easprintf**, **efopen**, **emalloc**, **ecalloc**, **erealloc**, **estrdup**, **estrndup**, **estrlcat**, **estrlcpy**, **evasprintf** — error-checked utility functions

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

void (*)(int, const char *, ...)
esetfunc(void (*)(int, const char *, ...));

int
easprintf(char ** restrict str, const char * restrict fmt, ...);

FILE *
efopen(const char *p, const char *m);

void *
ecalloc(size_t n, size_t c);

void *
emalloc(size_t n);

void *
erealloc(void *p, size_t n);

char *
estrdup(const char *s);

char *
estrndup(const char *s, size_t len);

size_t
estrlcat(char *dst, const char *src, size_t len);

size_t
estrlcpy(char *dst, const char *src, size_t len);

int
evasprintf(char ** restrict str, const char * restrict fmt, ...);
```

DESCRIPTION

The **easprintf**, **efopen**, **ecalloc**, **emalloc**, **erealloc**, **estrdup**, **estrndup**, **estrlcat**, **estrlcpy**, and **evasprintf** functions operate exactly as the corresponding functions that do not start with an ‘e’ except that in case of an error, they call the installed error handler that can be configured with **esetfunc**. For the string handling functions, it is an error when the destination buffer is not large enough to hold the complete string. For functions that allocate memory or open a file, it is an error when they would return a null pointer. The default error handler is `err`(3). The function **esetfunc** returns the previous error handler function. A `NULL` error handler will just call `exit`(3).

SEE ALSO

`asprintf`(3), `calloc`(3), `err`(3), `exit`(3), `fopen`(3), `malloc`(3), `realloc`(3), `strdup`(3), `strndup`(3), `strlcat`(3), `strlcpy`(3), `vasprintf`(3)

NAME

end, **etext**, **edata** — end boundaries of image segments

SYNOPSIS

```
extern int end;
extern int etext;
extern int edata;
```

DESCRIPTION

The globals *end*, *etext* and *edata* are program segment end addresses.

etext is located at the first address after the end of the text segment.

edata is located at the first address after the end of the initialized data segment.

end is located at the first address after the end of the data segment (BSS) when the program is loaded. Use the sbrk(2) system call with zero as its argument to find the current end of the data segment.

SEE ALSO

sbrk(2), malloc(3), a.out(5)

HISTORY

An **end** manual page appeared in Version 6 AT&T UNIX.

BUGS

Traditionally, no variable existed that pointed to the start of the text segment because the text segment always started at address zero. Although it is no longer valid to make this assumption, no variable similar to the ones documented above exists to point to the start of the text segment.

NAME

endutxent, **getutxent**, **getutxid**, **getutxline**, **pututxline**, **setutxent** — user accounting database functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <utmpx.h>

void
endutxent(void);

struct utmpx *
getutxent(void);

struct utmpx *
getutxid(const struct utmpx *);

struct utmpx *
getutxline(const struct utmpx *);

struct utmpx *
pututxline(const struct utmpx *);

void
setutxent(void);
```

DESCRIPTION

These functions provide access to the utmpx(5) user accounting database.

getutxent() reads the next entry from the database; if the database was not yet open, it also opens it. **setutxent**() resets the database, so that the next **getutxent**() call will get the first entry. **endutxent**() closes the database.

getutxid() returns the next entry of the type specified in its argument's *ut_type* field, or NULL if none is found. **getutxline**() returns the next LOGIN_PROCESS or USER_PROCESS entry which has the same name as specified in the *ut_line* field, or NULL if no match is found.

pututxline() adds the argument utmpx(5) entry line to the accounting database, replacing a previous entry for the same user if it exists.

The utmpx structure

The **utmpx** structure has the following definition:

```
struct utmpx {
        char ut_name[_UTX_USERSIZE];    /* login name */
        char ut_id[_UTX_IDSIZE];        /* inittab id */
        char ut_line[_UTX_LINESIZE];    /* tty name */
        char ut_host[_UTX_HOSTSIZE];    /* host name */
        uint16_t ut_session;            /* session id used for windowing */
        uint16_t ut_type;               /* type of this entry */
        pid_t ut_pid;                   /* process id creating the entry */
        struct {
                uint16_t e_termination; /* process termination signal */
                uint16_t e_exit;        /* process exit status */
        } ut_exit;
```

```
        struct sockaddr_storage ut_ss;  /* address where entry was made from */
        struct timeval ut_tv;           /* time entry was created */
        uint32_t ut_pad[10];            /* reserved for future use */
};
```

Valid entries for *ut_type* are:

BOOT_TIME	Time of a system boot.
DEAD_PROCESS	A session leader exited.
EMPTY	No valid user accounting information.
INIT_PROCESS	A process spawned by init(8).
LOGIN_PROCESS	The session leader of a logged-in user.
NEW_TIME	Time after system clock change.
OLD_TIME	Time before system clock change.
RUN_LVL	Run level. Provided for compatibility, not used on NetBSD.
USER_PROCESS	A user process.

RETURN VALUES

getutxent() returns the next entry, or NULL on failure (end of database or problems reading from the database). **getutxid**() and **getutxline**() return the matching structure on success, or NULL if no match was found. **pututxline**() returns the structure that was successfully written, or NULL.

SEE ALSO

logwtmpx(3), utmpx(5)

STANDARDS

The **endutxent**(), **getutxent**(), **getutxid**(), **getutxline**(), **pututxline**(), **setutxent**() all conform to IEEE Std 1003.1-2001 (“POSIX.1”) (XSI extension), and previously to X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”). The fields *ut_user*, *ut_id*, *ut_line*, *ut_pid*, *ut_type*, and *ut_tv* conform to IEEE Std 1003.1-2001 (“POSIX.1”) (XSI extension), and previously to X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”).

NAME

erf, **erff**, **erfc**, **erfcf** — error function operators

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
erf(double x);

float
erff(float x);

double
erfc(double x);

float
erfcf(float x);
```

DESCRIPTION

These functions calculate the error function of *x*.

The **erf**() calculates the error function of x; where

$$\mathrm{erf}(x) := (2/\sqrt{\pi}) \int_0^x \exp(-t^2)\, dt.$$

The **erfc**() function calculates the complementary error function of *x*; that is **erfc**() subtracts the result of the error function **erf**(*x*) from 1.0. This is useful, since for large *x* places disappear.

SEE ALSO

math(3)

HISTORY

The **erf**() and **erfc**() functions appeared in 4.3BSD.

NAME

err, **verr**, **errx**, **verrx**, **warn**, **vwarn**, **warnx**, **vwarnx** — formatted error messages

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <err.h>

void
err(int status, const char *fmt, ...);

void
verr(int status, const char *fmt, va_list args);

void
errx(int status, const char *fmt, ...);

void
verrx(int status, const char *fmt, va_list args);

void
warn(const char *fmt, ...);

void
vwarn(const char *fmt, va_list args);

void
warnx(const char *fmt, ...);

void
vwarnx(const char *fmt, va_list args);
```

DESCRIPTION

The **err**() and **warn**() family of functions display a formatted error message on the standard error output. In all cases, the last component of the program name, a colon character, and a space are output. If the *fmt* argument is not NULL, the formatted error message is output. In the case of the **err**(), **verr**(), **warn**(), and **vwarn**() functions, the error message string affiliated with the current value of the global variable *errno* is output next, preceded by a colon character and a space if *fmt* is not NULL. In all cases, the output is followed by a newline character. The **errx**(), **verrx**(), **warnx**(), and **vwarnx**() functions will not output this error message string.

The **err**(), **verr**(), **errx**(), and **verrx**() functions do not return, but instead cause the program to terminate with the status value given by the argument *status*. It is often appropriate to use the value EXIT_FAILURE, defined in ⟨*stdlib.h*⟩, as the *status* argument given to these functions.

EXAMPLES

Display the current *errno* information string and terminate with status indicating failure:

```
if ((p = malloc(size)) == NULL)
        err(EXIT_FAILURE, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
        err(EXIT_FAILURE, "%s", file_name);
```

Display an error message and terminate with status indicating failure:

```
if (tm.tm_hour < START_TIME)
        errx(EXIT_FAILURE, "too early, wait until %s",
            start_time_string);
```

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
        warnx("%s: %s: trying the block device",
            raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
        warn("%s", block_device);
```

SEE ALSO

`exit`(3), `getprogname`(3), `strerror`(3)

HISTORY

The **`err`**() and **`warn`**() functions first appeared in 4.4BSD.

CAVEATS

It is important never to pass a string with user-supplied data as a format without using '`%s`'. An attacker can put format specifiers in the string to mangle your stack, leading to a possible security hole. This holds true even if you have built the string "by hand" using a function like **`snprintf`**(), as the resulting string may still contain user-supplied conversion specifiers for later interpolation by the **`err`**() and **`warn`**() functions.

Always be sure to use the proper secure idiom:

```
err(1, "%s", string);
```

NAME

ether_ntoa, **ether_aton**, **ether_ntohost**, **ether_hostton**, **ether_line**, — get ethers entry

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <net/if_ether.h>

char *
ether_ntoa(const struct ether_addr *e);

struct ether_addr *
ether_aton(const char *s);

int
ether_ntohost(char *hostname, const struct ether_addr *e);

int
ether_hostton(const char *hostname, struct ether_addr *e);

int
ether_line(const char *line, struct ether_addr *e, char *hostname);
```

DESCRIPTION

Ethernet addresses are represented by the following structure:

```
struct ether_addr {
        u_char  ether_addr_octet[6];
};
```

The **ether_ntoa**() function converts this structure into an ASCII string of the form ‘‘xx:xx:xx:xx:xx:xx’’, consisting of 6 hexadecimal numbers separated by colons. It returns a pointer to a static buffer that is reused for each call. The **ether_aton**() converts an ASCII string of the same form and to a structure containing the 6 octets of the address. It returns a pointer to a static structure that is reused for each call.

The **ether_ntohost**() and **ether_hostton**() functions interrogate the data base mapping host names to Ethernet addresses, `/etc/ethers`. The **ether_ntohost**() function looks up the given Ethernet address and writes the associated host name into the character buffer passed. The **ether_hostton**() function looks up the given host name and writes the associated Ethernet address into the structure passed. Both functions return zero if they find the requested host name or address, and -1 if not. Each call reads `/etc/ethers` from the beginning; if a + appears alone on a line in the file, then **ether_hostton**() will consult the `ethers.byname` YP map, and **ether_ntohost**() will consult the `ethers.byaddr` YP map.

The **ether_line**() function parses a line from the `/etc/ethers` file and fills in the passed ‘‘struct ether_addr’’ and character buffer with the Ethernet address and host name on the line. It returns zero if the line was successfully parsed and -1 if not.

The *hostname* buffer for **ether_line**() and **ether_ntohost**() should be at least `MAXHOSTNAMELEN` + 1 characters long, to prevent a buffer overflow during parsing.

FILES

`/etc/ethers`

SEE ALSO

`ethers`(5)

HISTORY

The **`ether_ntoa`**(), **`ether_aton`**(), **`ether_ntohost`**(), **`ether_hostton`**(), and **`ether_line`**() functions were adopted from SunOS and appeared in NetBSD 1.0.

BUGS

The data space used by these functions is static; if future use requires the data, it should be copied before any subsequent calls to these functions overwrite it.

NAME

evdns_init **evdns_shutdown** **evdns_err_to_string** **evdns_nameserver_add** **evdns_count_nameservers** **evdns_clear_nameservers_and_suspend** **evdns_resume** **evdns_nameserver_ip_add** **evdns_resolve_ipv4** **evdns_resolve_reverse** **evdns_resolv_conf_parse** **evdns_search_clear** **evdns_search_add** **evdns_search_ndots_set** **evdns_set_log_fn** — asynchronous functions for DNS resolution

SYNOPSIS

```
#include <sys/time.h>
#include <event.h>
#include <evdns.h>

int
evdns_init();

void
evdns_shutdown(int fail_requests);

const char *
evdns_err_to_string(int err);

int
evdns_nameserver_add(unsigned long int address);

int
evdns_count_nameservers();

int
evdns_clear_nameservers_and_suspend();

int
evdns_resume();

int
evdns_nameserver_ip_add(const char *ip_as_string);

int
evdns_resolve_ipv4(const char *name, int flags,
    evdns_callback_type callback, void *ptr);

int
evdns_resolve_reverse(struct in_addr *in, int flags,
    evdns_callback_type callback, void *ptr);

int
evdns_resolv_conf_parse(int flags, const char *);

void
evdns_search_clear();

void
evdns_search_add(const char *domain);

void
evdns_search_ndots_set(const int ndots);

void
evdns_set_log_fn(evdns_debug_log_fn_type fn);
```

DESCRIPTION

Welcome, gentle reader

Async DNS lookups are really a whole lot harder than they should be, mostly stemming from the fact that the libc resolver has never been very good at them. Before you use this library you should see if libc can do the job for you with the modern async call getaddrinfo_a (see http://www.imperialviolet.org/page25.html#e498). Otherwise, please continue.

This code is based on libevent and you must call event_init before any of the APIs in this file. You must also seed the OpenSSL random source if you are using OpenSSL for ids (see below).

This library is designed to be included and shipped with your source code. You statically link with it. You should also test for the existence of strtok_r and define HAVE_STRTOK_R if you have it.

The DNS protocol requires a good source of id numbers and these numbers should be unpredictable for spoofing reasons. There are three methods for generating them here and you must define exactly one of them. In increasing order of preference:

DNS_USE_GETTIMEOFDAY_FOR_ID
: Using the bottom 16 bits of the usec result from gettimeofday. This is a pretty poor solution but should work anywhere.

DNS_USE_CPU_CLOCK_FOR_ID
: Using the bottom 16 bits of the nsec result from the CPU's time counter. This is better, but may not work everywhere. Requires POSIX realtime support and you'll need to link against -lrt on glibc systems at least.

DNS_USE_OPENSSL_FOR_ID
: Uses the OpenSSL RAND_bytes call to generate the data. You must have seeded the pool before making any calls to this library.

The library keeps track of the state of nameservers and will avoid them when they go down. Otherwise it will round robin between them.

Quick start guide:

```
#include "evdns.h"
void callback(int result, char type, int count, int ttl,
    void *addresses, void *arg);
evdns_resolv_conf_parse(DNS_OPTIONS_ALL, "/etc/resolv.conf");
evdns_resolve("www.hostname.com", 0, callback, NULL);
```

When the lookup is complete the callback function is called. The first argument will be one of the DNS_ERR_* defines in evdns.h. Hopefully it will be DNS_ERR_NONE, in which case type will be DNS_IPv4_A, count will be the number of IP addresses, ttl is the time which the data can be cached for (in seconds), addresses will point to an array of uint32_t's and arg will be whatever you passed to evdns_resolve.

Searching:

In order for this library to be a good replacement for glibc's resolver it supports searching. This involves setting a list of default domains, in which names will be queried for. The number of dots in the query name determines the order in which this list is used.

Searching appears to be a single lookup from the point of view of the API, although many DNS queries may be generated from a single call to evdns_resolve. Searching can also drastically slow down the resolution of names.

To disable searching:

1. Never set it up. If you never call **evdns_resolv_conf_parse**(), **evdns_init**(), or **evdns_search_add**() then no searching will occur.
2. If you do call **evdns_resolv_conf_parse**() then don't pass *DNS_OPTION_SEARCH* (or *DNS_OPTIONS_ALL*, which implies it).
3. When calling **evdns_resolve**(), pass the *DNS_QUERY_NO_SEARCH* flag.

The order of searches depends on the number of dots in the name. If the number is greater than the ndots setting then the names is first tried globally. Otherwise each search domain is appended in turn.

The ndots setting can either be set from a resolv.conf, or by calling evdns_search_ndots_set.

For example, with ndots set to 1 (the default) and a search domain list of ["myhome.net"]:
Query: www
Order: www.myhome.net, www.

Query: www.abc
Order: www.abc., www.abc.myhome.net

API reference

int **evdns_init**()
Initializes support for non-blocking name resolution by calling **evdns_resolv_conf_parse**().

int **evdns_nameserver_add**(*unsigned long int address*)
Add a nameserver. The address should be an IP address in network byte order. The type of address is chosen so that it matches in_addr.s_addr. Returns non-zero on error.

int **evdns_nameserver_ip_add**(*const char *ip_as_string*)
This wraps the above function by parsing a string as an IP address and adds it as a nameserver. Returns non-zero on error

int **evdns_resolve**(*const char *name*, *int flags*, *evdns_callback_type callback*, *void *ptr*)
Resolve a name. The name parameter should be a DNS name. The flags parameter should be 0, or DNS_QUERY_NO_SEARCH which disables searching for this query. (see defn of searching above).

The callback argument is a function which is called when this query completes and ptr is an argument which is passed to that callback function.

Returns non-zero on error

void **evdns_search_clear**()
Clears the list of search domains

void **evdns_search_add**(*const char *domain*)
Add a domain to the list of search domains

void **evdns_search_ndots_set**(*int ndots*)
Set the number of dots which, when found in a name, causes the first query to be without any search domain.

int **evdns_count_nameservers**(*void*)
Return the number of configured nameservers (not necessarily the number of running nameservers). This is useful for double-checking whether our calls to the various nameserver configuration functions have been successful.

int **evdns_clear_nameservers_and_suspend**(*void*)
Remove all currently configured nameservers, and suspend all pending resolves. Resolves will not necessarily be re-attempted until **evdns_resume**() is called.

int **evdns_resume**(*void*)
Re-attempt resolves left in limbo after an earlier call to **evdns_clear_nameservers_and_suspend**().

int **evdns_resolv_conf_parse**(*int flags*, *const char *filename*)
Parse a resolv.conf like file from the given filename.

See the man page for resolv.conf for the format of this file. The flags argument determines what information is parsed from this file:

DNS_OPTION_SEARCH	domain, search and ndots options
DNS_OPTION_NAMESERVERS	nameserver lines
DNS_OPTION_MISC	timeout and attempts options
DNS_OPTIONS_ALL	all of the above

The following directives are not parsed from the file:
sortlist, rotate, no-check-names, inet6, debug

Returns non-zero on error:

0	no errors
1	failed to open file
2	failed to stat file
3	file too large
4	out of memory
5	short read from file

Internals

Requests are kept in two queues. The first is the inflight queue. In this queue requests have an allocated transaction id and nameserver. They will soon be transmitted if they haven't already been.

The second is the waiting queue. The size of the inflight ring is limited and all other requests wait in waiting queue for space. This bounds the number of concurrent requests so that we don't flood the nameserver. Several algorithms require a full walk of the inflight queue and so bounding its size keeps thing going nicely under huge (many thousands of requests) loads.

If a nameserver loses too many requests it is considered down and we try not to use it. After a while we send a probe to that nameserver (a lookup for google.com) and, if it replies, we consider it working again. If the nameserver fails a probe we wait longer to try again with the next probe.

SEE ALSO

event(3), gethostbyname(3), resolv.conf(5)

HISTORY

The **evdns** API was developed by Adam Langley on top of the **libevent** API. The code was integrate into **Tor** by Nick Mathewson and finally put into **libevent** itself by Niels Provos.

AUTHORS

The **evdns** API and code was written by Adam Langley with significant contributions by Nick Mathewson.

BUGS

This documentation is neither complete nor authoritative. If you are in doubt about the usage of this API then check the source code to find out how it works, write up the missing piece of documentation and send it

to me for inclusion in this man page.

NAME

event_init, **event_dispatch**, **event_loop**, **event_loopexit**, **event_loopbreak**, **event_set**, **event_base_dispatch**, **event_base_loop**, **event_base_loopexit**, **event_base_loopbreak**, **event_base_set**, **event_base_free**, **event_add**, **event_del**, **event_once**, **event_base_once**, **event_pending**, **event_initialized**, **event_priority_init**, **event_priority_set**, **evtimer_set**, **evtimer_add**, **evtimer_del**, **evtimer_pending**, **evtimer_initialized**, **signal_set**, **signal_add**, **signal_del**, **signal_pending**, **signal_initialized**, **bufferevent_new**, **bufferevent_free**, **bufferevent_write**, **bufferevent_write_buffer**, **bufferevent_read**, **bufferevent_enable**, **bufferevent_disable**, **bufferevent_settimeout**, **bufferevent_base_set**, **evbuffer_new**, **evbuffer_free**, **evbuffer_add**, **evbuffer_add_buffer**, **evbuffer_add_printf**, **evbuffer_add_vprintf**, **evbuffer_drain**, **evbuffer_write**, **evbuffer_read**, **evbuffer_find**, **evbuffer_readline**, **evhttp_new**, **evhttp_bind_socket**, **evhttp_free** — execute a function when a specific event occurs

LIBRARY

Event Notification Library (libevent, −levent)

SYNOPSIS

```
#include <sys/time.h>
#include <event.h>

struct event_base *
event_init();

int
event_dispatch();

int
event_loop(int flags);

int
event_loopexit(struct timeval *tv);

int
event_loopbreak(void);

int
event_base_dispatch(struct event_base *base);

void
event_base_free(struct event_base *base);

int
event_base_loop(struct event_base *base, int flags);

int
event_base_loopexit(struct event_base *base, struct timeval *tv);

int
event_base_loopbreak(struct event_base *base);

int
event_base_set(struct event_base *base, struct event *);
```

void
event_set(*struct event *ev*, *int fd*, *short event*,
*void (*fn)(int, short, void *)*, *void *arg*);

int
event_add(*struct event *ev*, *struct timeval *tv*);

int
event_del(*struct event *ev*);

int
event_once(*int fd*, *short event*, *void (*fn)(int, short, void *)*, *void *arg*,
*struct timeval *tv*);

inr
event_base_once(*struct event_base *base*, *int fd*, *short event*,
*void (*fn)(int, short, void *)*, *void *arg*, *struct timeval *tv*);

int
event_pending(*struct event *ev*, *short event*, *struct timeval *tv*);

int
event_initialized(*struct event *ev*);

void
evtimer_set(*struct event *ev*, *void (*fn)(int, short, void *)*, *void *arg*);

void
evtimer_add(*struct event *ev*, *struct timeval *tv*);

void
evtimer_del(*struct event *ev*);

int
evtimer_pending(*struct event *ev*, *struct timeval *tv*);

int
evtimer_initialized(*struct event *ev*);

void
signal_set(*struct event *ev*, *int signal*, *void (*fn)(int, short, void *)*,
*void *arg*);

void
signal_add(*struct event *ev*, *struct timeval *tv*);

void
signal_del(*struct event *ev*);

int
signal_pending(*struct event *ev*, *struct timeval *tv*);

int
signal_initialized(*struct event *ev*);

*struct bufferevent **
bufferevent_new(*int fd*, *evbuffercb readcb*, *evbuffercb writecb*, *everrorcb*,
*void *cbarg*);

void
bufferevent_free(*struct bufferevent *bufev*);

int
bufferevent_write(*struct bufferevent *bufev*, *void *data*, *size_t size*);

int
bufferevent_write_buffer(*struct bufferevent *bufev*, *struct evbuffer *buf*);

size_t
bufferevent_read(*struct bufferevent *bufev*, *void *data*, *size_t size*);

int
bufferevent_enable(*struct bufferevent *bufev*, *short event*);

int
bufferevent_disable(*struct bufferevent *bufev*, *short event*);

void
bufferevent_settimeout(*struct bufferevent *bufev*, *int timeout_read*,
int timeout_write);

int
bufferevent_base_set(*struct event_base *base*, *struct bufferevent *bufev*);

*struct evbuffer **
evbuffer_new(*void*);

void
evbuffer_free(*struct evbuffer *buf*);

int
evbuffer_add(*struct evbuffer *buf*, *const void *data*, *size_t size*);

int
evbuffer_add_buffer(*struct evbuffer *dst*, *struct evbuffer *src*);

int
evbuffer_add_printf(*struct evbuffer *buf*, *const char *fmt*, *...*);

int
evbuffer_add_vprintf(*struct evbuffer *buf*, *const char *fmt*, *va_list ap*);

void
evbuffer_drain(*struct evbuffer *buf*, *size_t size*);

int
evbuffer_write(*struct evbuffer *buf*, *int fd*);

int
evbuffer_read(*struct evbuffer *buf*, *int fd*, *int size*);

*u_char **
evbuffer_find(*struct evbuffer *buf*, *const u_char *data*, *size_t size*);

*char **
evbuffer_readline(*struct evbuffer *buf*);

*struct evhttp **
evhttp_new(*struct event_base *base*);

int
evhttp_bind_socket(*struct evhttp *http*, *const char *address*, *u_short port*);

void
evhttp_free(*struct evhttp *http*);

*int (*event_sigcb)(void)*;

volatile sig_atomic_t event_gotsig;

DESCRIPTION

The **event** API provides a mechanism to execute a function when a specific event on a file descriptor occurs or after a given time has passed.

The **event** API needs to be initialized with **event_init**() before it can be used.

In order to process events, an application needs to call **event_dispatch**(). This function only returns on error, and should replace the event core of the application program.

The function **event_set**() prepares the event structure *ev* to be used in future calls to **event_add**() and **event_del**(). The event will be prepared to call the function specified by the *fn* argument with an *int* argument indicating the file descriptor, a *short* argument indicating the type of event, and a *void* * argument given in the *arg* argument. The *fd* indicates the file descriptor that should be monitored for events. The events can be either *EV_READ*, *EV_WRITE*, or both, indicating that an application can read or write from the file descriptor respectively without blocking.

The function *fn* will be called with the file descriptor that triggered the event and the type of event which will be either *EV_TIMEOUT*, *EV_SIGNAL*, *EV_READ*, or *EV_WRITE*. Additionally, an event which has registered interest in more than one of the preceeding events, via bitwise-OR to **event_set**(), can provide its callback function with a bitwise-OR of more than one triggered event. The additional flag *EV_PERSIST* makes an **event_add**() persistent until **event_del**() has been called.

Once initialized, the *ev* structure can be used repeatedly with **event_add**() and **event_del**() and does not need to be reinitialized unless the function called and/or the argument to it are to be changed. However, when an *ev* structure has been added to libevent using **event_add**() the structure must persist until the event occurs (assuming *EV_PERSIST* is not set) or is removed using **event_del**(). You may not reuse the same *ev* structure for multiple monitored descriptors; each descriptor needs its own *ev*.

The function **event_add**() schedules the execution of the *ev* event when the event specified in **event_set**() occurs or in at least the time specified in the *tv*. If *tv* is NULL, no timeout occurs and the function will only be called if a matching event occurs on the file descriptor. The event in the *ev* argument must be already initialized by **event_set**() and may not be used in calls to **event_set**() until it has timed out or been removed with **event_del**(). If the event in the *ev* argument already has a scheduled timeout, the old timeout will be replaced by the new one.

The function **event_del**() will cancel the event in the argument *ev*. If the event has already executed or has never been added the call will have no effect.

The functions **evtimer_set**(), **evtimer_add**(), **evtimer_del**(), **evtimer_initialized**(), and **evtimer_pending**() are abbreviations for common situations where only a timeout is required. The file descriptor passed will be −1, and the event type will be *EV_TIMEOUT*.

The functions **signal_set**(), **signal_add**(), **signal_del**(), **signal_initialized**(), and **signal_pending**() are abbreviations. The event type will be a persistent *EV_SIGNAL*. That means **signal_set**() adds *EV_PERSIST*.

In order to avoid races in signal handlers, the **event** API provides two variables: *event_sigcb* and *event_gotsig*. A signal handler sets *event_gotsig* to indicate that a signal has been received. The application

sets *event_sigcb* to a callback function. After the signal handler sets *event_gotsig*, **event_dispatch** will execute the callback function to process received signals. The callback returns 1 when no events are registered any more. It can return −1 to indicate an error to the **event** library, causing **event_dispatch**() to terminate with *errno* set to EINTR.

The function **event_once**() is similar to **event_set**(). However, it schedules a callback to be called exactly once and does not require the caller to prepare an *event* structure. This function supports *EV_TIMEOUT*, *EV_READ*, and *EV_WRITE*.

The **event_pending**() function can be used to check if the event specified by *event* is pending to run. If *EV_TIMEOUT* was specified and *tv* is not NULL, the expiration time of the event will be returned in *tv*.

The **event_initialized**() macro can be used to check if an event has been initialized.

The **event_loop** function provides an interface for single pass execution of pending events. The flags *EVLOOP_ONCE* and *EVLOOP_NONBLOCK* are recognized. The **event_loopexit** function exits from the event loop. The next **event_loop**() iteration after the given timer expires will complete normally (handling all queued events) then exit without blocking for events again. Subsequent invocations of **event_loop**() will proceed normally. The **event_loopbreak** function exits from the event loop immediately. **event_loop**() will abort after the next event is completed; **event_loopbreak**() is typically invoked from this event's callback. This behavior is analogous to the "break;" statement. Subsequent invocations of **event_loop**() will proceed normally.

It is the responsibility of the caller to provide these functions with pre-allocated event structures.

EVENT PRIORITIES

By default **libevent** schedules all active events with the same priority. However, sometimes it is desirable to process some events with a higher priority than others. For that reason, **libevent** supports strict priority queues. Active events with a lower priority are always processed before events with a higher priority.

The number of different priorities can be set initially with the **event_priority_init**() function. This function should be called before the first call to **event_dispatch**(). The **event_priority_set**() function can be used to assign a priority to an event. By default, **libevent** assigns the middle priority to all events unless their priority is explicitly set.

THREAD SAFE EVENTS

Libevent has experimental support for thread-safe events. When initializing the library via **event_init**(), an event base is returned. This event base can be used in conjunction with calls to **event_base_set**(), **event_base_dispatch**(), **event_base_loop**(), **event_base_loopexit**(), **bufferevent_base_set**() and **event_base_free**(). **event_base_set**() should be called after preparing an event with **event_set**(), as **event_set**() assigns the provided event to the most recently created event base. **bufferevent_base_set**() should be called after preparing a bufferevent with **bufferevent_new**(). **event_base_free**() should be used to free memory associated with the event base when it is no longer needed.

BUFFERED EVENTS

libevent provides an abstraction on top of the regular event callbacks. This abstraction is called a "**buffered event**". A buffered event provides input and output buffers that get filled and drained automatically. The user of a buffered event no longer deals directly with the IO, but instead is reading from input and writing to output buffers.

A new bufferevent is created by **bufferevent_new**(). The parameter *fd* specifies the file descriptor from which data is read and written to. This file descriptor is not allowed to be a pipe(2). The next three parameters are callbacks. The read and write callback have the following form: *void* **(**∗**cb)**(*struct bufferevent* ∗*bufev*, *void* ∗*arg*). The error callback has the following form: *void*

(*cb)(*struct bufferevent *bufev*, *short what*, *void *arg*). The argument is specified by the fourth parameter *cbarg*. A *bufferevent struct* pointer is returned on success, NULL on error. Both the read and the write callback may be NULL. The error callback has to be always provided.

Once initialized, the bufferevent structure can be used repeatedly with **bufferevent_enable**() and **bufferevent_disable**(). The flags parameter can be a combination of *EV_READ* and *EV_WRITE*. When read enabled the bufferevent will try to read from the file descriptor and call the read callback. The write callback is executed whenever the output buffer is drained below the write low watermark, which is *0* by default.

The **bufferevent_write**() function can be used to write data to the file descriptor. The data is appended to the output buffer and written to the descriptor automatically as it becomes available for writing. **bufferevent_write**() returns 0 on success or −1 on failure. The **bufferevent_read**() function is used to read data from the input buffer, returning the amount of data read.

If multiple bases are in use, **bufferevent_base_set**() must be called before enabling the bufferevent for the first time.

NON-BLOCKING HTTP SUPPORT

libevent provides a very thin HTTP layer that can be used both to host an HTTP server and also to make HTTP requests. An HTTP server can be created by calling **evhttp_new**(). It can be bound to any port and address with the **evhttp_bind_socket**() function. When the HTTP server is no longer used, it can be freed via **evhttp_free**().

To be notified of HTTP requests, a user needs to register callbacks with the HTTP server. This can be done by calling **evhttp_set_cb**(). The second argument is the URI for which a callback is being registered. The corresponding callback will receive an *struct evhttp_request* object that contains all information about the request.

This section does not document all the possible function calls; please check *event.h* for the public interfaces.

ADDITIONAL NOTES

It is possible to disable support for *epoll*, *kqueue*, *devpoll*, *poll* or *select* by setting the environment variable *EVENT_NOEPOLL*, *EVENT_NOKQUEUE*, *EVENT_NODEVPOLL*, *EVENT_NOPOLL* or *EVENT_NOSELECT*, respectively. By setting the environment variable *EVENT_SHOW_METHOD*, **libevent** displays the kernel notification method that it uses.

RETURN VALUES

Upon successful completion **event_add**() and **event_del**() return 0. Otherwise, −1 is returned and the global variable errno is set to indicate the error.

SEE ALSO

kqueue(2), poll(2), evdns(3), timeout(9)

HISTORY

event_init appeared in NetBSD 2.0. The **event** API manpage is based on the timeout(9) manpage by Artur Grabowski.

AUTHORS

The **event** library was written by Niels Provos.

BUGS

This documentation is neither complete nor authoritative. If you are in doubt about the usage of this API then check the source code to find out how it works, write up the missing piece of documentation and send it

to me for inclusion in this man page.

NAME

execl, **execlp**, **execle**, **exect**, **execv**, **execvp** — execute a file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

extern char **environ;

int
execl(const char *path, const char *arg, ...);

int
execlp(const char *file, const char *arg, ...);

int
execle(const char *path, const char *arg, ..., char *const envp[]);

int
exect(const char *path, char *const argv[], char *const envp[]);

int
execv(const char *path, char *const argv[]);

int
execvp(const char *file, char *const argv[]);
```

DESCRIPTION

The **exec** family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function execve(2). (See the manual page for execve(2) for detailed information about the replacement of the current process. The script(7) manual page provides detailed information about the execution of interpreter scripts.)

The initial argument for these functions is the pathname of a file which is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer.

The **exect**(), **execv**(), and **execvp**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers **must** be terminated by a NULL pointer.

The **execle**() and **exect**() functions also specify the environment of the executed process by following the NULL pointer that terminates the list of arguments in the parameter list or the pointer to the argv array with an additional parameter. This additional parameter is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the current process.

Some of these functions have special semantics.

The functions **execlp**() and **execvp**() will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash "/" character. The search path is the path specified in

the environment by the PATH variable. If this variable isn't specified, *_PATH_DEFPATH* from ⟨paths.h⟩ is used instead, its value being: /usr/bin:/bin:/usr/pkg/bin:/usr/local/bin. In addition, certain errors are treated specially.

If permission is denied for a file (the attempted execve(2) returned EACCES), these functions will continue searching the rest of the search path. If no other file is found, however, they will return with the global variable *errno* set to EACCES.

If the header of a file isn't recognized (the attempted execve(2) returned ENOEXEC), these functions will execute the shell with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

If the file is currently busy (the attempted execve(2) returned ETXTBUSY), these functions will sleep for several seconds, periodically re-attempting to execute the file.

The function **exect**() executes a file with the program tracing facilities enabled (see ptrace(2)).

RETURN VALUES

If any of the **exec** functions returns, an error will have occurred. The return value is −1, and the global variable *errno* will be set to indicate the error.

FILES

/bin/sh The shell.

ERRORS

execl(), **execle**(), **execlp**() and **execvp**() may fail and set *errno* for any of the errors specified for the library functions execve(2) and malloc(3).

exect() and **execv**() may fail and set *errno* for any of the errors specified for the library function execve(2).

SEE ALSO

sh(1), execve(2), fork(2), ptrace(2), environ(7), script(7)

COMPATIBILITY

Historically, the default path for the **execlp**() and **execvp**() functions was ":/bin:/usr/bin". This was changed to improve security and behaviour.

The behavior of **execlp**() and **execvp**() when errors occur while attempting to execute the file is historic practice, but has not traditionally been documented and is not specified by the POSIX standard.

Traditionally, the functions **execlp**() and **execvp**() ignored all errors except for the ones described above and ENOMEM and E2BIG, upon which they returned. They now return if any error other than the ones described above occurs.

STANDARDS

execl(), **execv**(), **execle**(), **execlp**() and **execvp**() conform to ISO/IEC 9945-1:1990 ("POSIX.1").

NAME

exit — perform normal program termination

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

void
exit(int status);
```

DESCRIPTION

exit() terminates a process. The *status* values EXIT_SUCCESS and EXIT_FAILURE can be used to indicate successful and unsuccessful termination, respectively.

Before termination it performs the following functions in the order listed:

1. Call the functions registered with the atexit(3) function, in the reverse order of their registration.
2. Flush all open output streams.
3. Close all open streams.
4. Unlink all files created with the tmpfile(3) function.

Following this, **exit**() calls _exit(2). Note that typically _exit(2) only passes the lower 8 bits of *status* on to the parent, thus negative values have less meaning.

RETURN VALUES

The **exit**() function never returns.

SEE ALSO

_exit(2), atexit(3), intro(3), tmpfile(3)

STANDARDS

The **exit**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

exp, **expf**, **expm1**, **expm1f**, **log**, **logf**, **log2**, **log2f**, **log10**, **log10f**, **log1p**, **log1pf**, **pow**, **powf** — exponential, logarithm, power functions

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
exp(double x);

float
expf(float x);

double
expm1(double x);

float
expm1f(float x);

double
log(double x);

float
logf(float x);

double
log2(double x);

float
log2f(float x);

double
log10(double x);

float
log10f(float x);

double
log1p(double x);

float
log1pf(float x);

double
pow(double x, double y);

float
powf(float x, float y);
```

DESCRIPTION

The **exp**() function computes the exponential value of the given argument *x*.

The **expm1**() function computes the value exp(x)−1 accurately even for tiny argument *x*.

The **log**() function computes the value of the natural logarithm of argument *x*.

The **log10**() function computes the value of the logarithm of argument *x* to base 10.

The **log1p**() function computes the value of log(1+x) accurately even for tiny argument *x*.

The **log2**() and the **log2f**() functions compute the value of the logarithm of argument *x* to base 2.

The **pow**() computes the value of *x* to the exponent *y*.

RETURN VALUES

These functions will return the appropriate computation unless an error occurs or an argument is out of range. The functions **exp**(), **expm1**() and **pow**() detect if the computed value will overflow, set the global variable *errno* to ERANGE and cause a reserved operand fault on a VAX. The function **pow**(*x*, *y*) checks to see if *x* < 0 and *y* is not an integer, in the event this is true, the global variable *errno* is set to EDOM and on the VAX generate a reserved operand fault. On a VAX, *errno* is set to EDOM and the reserved operand is returned by log unless *x* > 0, by **log1p**() unless *x* > −1.

ERRORS

exp(x), log(x), expm1(x) and log1p(x) are accurate to within an *ulp*, and log10(x) to within about 2 *ulps*; an *ulp* is one *Unit* in the *Last Place*. The error in **pow**(*x*, *y*) is below about 2 *ulps* when its magnitude is moderate, but increases as **pow**(*x*, *y*) approaches the over/underflow thresholds until almost as many bits could be lost as are occupied by the floating−point format's exponent field; that is 8 bits for VAX D and 11 bits for IEEE 754 Double. No such drastic loss has been exposed by testing; the worst errors observed have been below 20 *ulps* for VAX D, 300 *ulps* for IEEE 754 Double. Moderate values of **pow**() are accurate enough that **pow**(*integer*, *integer*) is exact until it is bigger than 2**56 on a VAX, 2**53 for IEEE 754.

NOTES

The functions exp(x)−1 and log(1+x) are called expm1 and logp1 in BASIC on the Hewlett−Packard HP−71B and APPLE Macintosh, EXP1 and LN1 in Pascal, exp1 and log1 in C on APPLE Macintoshes, where they have been provided to make sure financial calculations of ((1+x)**n−1)/x, namely expm1(n*log1p(x))/x, will be accurate when x is tiny. They also provide accurate inverse hyperbolic functions.

The function **pow**(*x*, *0*) returns x**0 = 1 for all x including x = 0, ∞ (not found on a VAX), and *NaN* (the reserved operand on a VAX). Previous implementations of pow may have defined x**0 to be undefined in some or all of these cases. Here are reasons for returning x**0 = 1 always:

1. Any program that already tests whether x is zero (or infinite or *NaN*) before computing x**0 cannot care whether 0**0 = 1 or not. Any program that depends upon 0**0 to be invalid is dubious anyway since that expression's meaning and, if invalid, its consequences vary from one computer system to another.

2. Some Algebra texts (e.g. Sigler's) define x**0 = 1 for all x, including x = 0. This is compatible with the convention that accepts a[0] as the value of polynomial

   ```
   p(x) = a[0]*x**0 + a[1]*x**1 + a[2]*x**2 +...+ a[n]*x**n
   ```

   at x = 0 rather than reject a[0]*0**0 as invalid.

3. Analysts will accept 0**0 = 1 despite that x**y can approach anything or nothing as x and y approach 0 independently. The reason for setting 0**0 = 1 anyway is this:

   > If x(z) and y(z) are *any* functions analytic (expandable in power series) in z around z = 0, and if there x(0) = y(0) = 0, then x(z)**y(z) → 1 as z → 0.

4. If 0**0 = 1, then ∞**0 = 1/0**0 = 1 too; and then *NaN***0 = 1 too because x**0 = 1 for all finite and infinite x, i.e., independently of x.

SEE ALSO

`math`(3)

STANDARDS

The **exp**(), **log**(), **log10**() and **pow**() functions conform to ANSI X3.159-1989 ("ANSI C89").

HISTORY

A **exp**(), **log**() and **pow**() functions appeared in Version 6 AT&T UNIX. A **log10**() function appeared in Version 7 AT&T UNIX. The **log1p**() and **expm1**() functions appeared in 4.3BSD.

NAME

extattr_namespace_to_string, **extattr_string_to_namespace** — convert an extended attribute namespace identifier to a string and vice versa

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/extattr.h>

int
extattr_namespace_to_string(int attrnamespace, char **string);

int
extattr_string_to_namespace(const char *string, int *attrnamespace);
```

DESCRIPTION

The **extattr_namespace_to_string**() function converts a VFS extended attribute identifier to a human-readable string. The **extattr_string_to_namespace**() converts a human-readable string representing a namespace to a namespace identifier. Although a file system may implement arbitrary namespaces, these functions only support the EXTATTR_NAMESPACE_USER ("user") and EXTATTR_NAMESPACE_SYSTEM ("system") namespaces, which are defined in extattr(9).

These functions are meant to be used in error reporting and other interactive tasks. For example, instead of printing the integer identifying an extended attribute in an error message, a program might use **extattr_namespace_to_string**() to obtain a human-readable representation. Likewise, instead of requiring a user to enter the integer representing a namespace, an interactive program might ask for a name and use **extattr_string_to_namespace**() to get the desired identifier.

RETURN VALUES

If any of the calls are unsuccessful, the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] The requested namespace could not be identified.

SEE ALSO

getextattr(1), extattr_get_file(2), extattr(9)

HISTORY

Extended attribute support was developed as part of the TrustedBSD Project, and introduced in FreeBSD 5.0 and NetBSD 3.0. It was developed to support security extensions requiring additional labels to be associated with each file or directory.

NAME

fabs, **fabsf** — floating-point absolute value function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
fabs(double x);

float
fabsf(float x);
```

DESCRIPTION

The **fabs**() and **fabsf**() functions compute the absolute value of a floating-point number *x*.

RETURN VALUES

The **fabs**() function returns the absolute value of *x*.

SEE ALSO

abs(3), ceil(3), floor(3), ieee(3), math(3), rint(3)

STANDARDS

The **fabs**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

fclose — close a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

int
fclose(FILE *stream);
```

DESCRIPTION

The **fclose**() function dissociates the named *stream* from its underlying file or set of functions. If the stream was being used for output, any buffered data is written first, using fflush(3).

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, EOF is returned and the global variable *errno* is set to indicate the error. In either case no further access to the stream is possible.

ERRORS

[EBADF] The argument *stream* is not an open stream.

The **fclose**() function may also fail and set *errno* for any of the errors specified for the routines close(2) or fflush(3).

SEE ALSO

close(2), fflush(3), fopen(3), setbuf(3)

STANDARDS

The **fclose**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

clearerr, **feof**, **ferror**, **fileno** — check and reset stream status

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

void
clearerr(FILE *stream);

int
feof(FILE *stream);

int
ferror(FILE *stream);

int
fileno(FILE *stream);
```

DESCRIPTION

The function **clearerr**() clears the end-of-file and error indicators for the stream pointed to by *stream*.

The function **feof**() tests the end-of-file indicator for the stream pointed to by *stream*, returning non-zero if it is set. The end-of-file indicator can only be cleared by the function **clearerr**().

The function **ferror**() tests the error indicator for the stream pointed to by *stream*, returning non-zero if it is set. The error indicator can only be reset by the **clearerr**() function.

The function **fileno**() examines the argument *stream* and returns its integer descriptor.

ERRORS

These functions should not fail and do not set the external variable *errno*.

SEE ALSO

open(2), stdio(3)

STANDARDS

The functions **clearerr**(), **feof**(), and **ferror**() conform to ANSI X3.159-1989 (“ANSI C89”). The function **fileno**() conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

fetchMakeURL, **fetchParseURL**, **fetchFreeURL**, **fetchXGetURL**, **fetchGetURL**, **fetchPutURL**, **fetchStatURL**, **fetchListURL**, **fetchXGet**, **fetchGet**, **fetchPut**, **fetchStat**, **fetchList**, **fetchXGetFile**, **fetchGetFile**, **fetchPutFile**, **fetchStatFile**, **fetchListFile**, **fetchXGetHTTP**, **fetchGetHTTP**, **fetchPutHTTP**, **fetchStatHTTP**, **fetchListHTTP**, **fetchXGetFTP**, **fetchGetFTP**, **fetchPutFTP**, **fetchStatFTP**, **fetchListFTP** — file transfer functions

LIBRARY

library “libfetch”

SYNOPSIS

```
#include <sys/param.h>
#include <stdio.h>
#include <fetch.h>

struct url *
fetchMakeURL(const char *scheme, const char *host, int port,
    const char *doc, const char *user, const char *pwd);

struct url *
fetchParseURL(const char *URL);

void
fetchFreeURL(struct url *u);

FILE *
fetchXGetURL(const char *URL, struct url_stat *us, const char *flags);

FILE *
fetchGetURL(const char *URL, const char *flags);

FILE *
fetchPutURL(const char *URL, const char *flags);

int
fetchStatURL(const char *URL, struct url_stat *us, const char *flags);

struct url_ent *
fetchListURL(const char *URL, const char *flags);

FILE *
fetchXGet(struct url *u, struct url_stat *us, const char *flags);

FILE *
fetchGet(struct url *u, const char *flags);

FILE *
fetchPut(struct url *u, const char *flags);

int
fetchStat(struct url *u, struct url_stat *us, const char *flags);

struct url_ent *
fetchList(struct url *u, const char *flags);

FILE *
fetchXGetFile(struct url *u, struct url_stat *us, const char *flags);
```

*FILE **
fetchGetFile(*struct url *u*, *const char *flags*);

*FILE **
fetchPutFile(*struct url *u*, *const char *flags*);

int
fetchStatFile(*struct url *u*, *struct url_stat *us*, *const char *flags*);

*struct url_ent **
fetchListFile(*struct url *u*, *const char *flags*);

*FILE **
fetchXGetHTTP(*struct url *u*, *struct url_stat *us*, *const char *flags*);

*FILE **
fetchGetHTTP(*struct url *u*, *const char *flags*);

*FILE **
fetchPutHTTP(*struct url *u*, *const char *flags*);

int
fetchStatHTTP(*struct url *u*, *struct url_stat *us*, *const char *flags*);

*struct url_ent **
fetchListHTTP(*struct url *u*, *const char *flags*);

*FILE **
fetchXGetFTP(*struct url *u*, *struct url_stat *us*, *const char *flags*);

*FILE **
fetchGetFTP(*struct url *u*, *const char *flags*);

*FILE **
fetchPutFTP(*struct url *u*, *const char *flags*);

int
fetchStatFTP(*struct url *u*, *struct url_stat *us*, *const char *flags*);

*struct url_ent **
fetchListFTP(*struct url *u*, *const char *flags*);

DESCRIPTION

These functions implement a high-level library for retrieving and uploading files using Uniform Resource Locators (URLs).

fetchParseURL() takes a URL in the form of a null-terminated string and splits it into its components function according to the Common Internet Scheme Syntax detailed in RFC1738. A regular expression which produces this syntax is:

```
<scheme>:(//(<user>(:<pwd>)?@)?<host>(:<port>)?)?/(<document>)?
```

If the URL does not seem to begin with a scheme name, the following syntax is assumed:

```
((<user>(:<pwd>)?@)?<host>(:<port>)?)?/(<document>)?
```

Note that some components of the URL are not necessarily relevant to all URL schemes. For instance, the file scheme only needs the <scheme> and <document> components.

fetchMakeURL() and **fetchParseURL**() return a pointer to a *url* structure, which is defined as follows in <fetch.h>:

```
#define URL_SCHEMELEN 16
#define URL_USERLEN 256
#define URL_PWDLEN 256

struct url {
    char        scheme[URL_SCHEMELEN+1];
    char        user[URL_USERLEN+1];
    char        pwd[URL_PWDLEN+1];
    char        host[MAXHOSTNAMELEN+1];
    int         port;
    char       *doc;
    off_t       offset;
    size_t      length;
};
```

The pointer returned by **fetchMakeURL**() or **fetchParseURL**() should be freed using **fetchFreeURL**().

fetchXGetURL(), **fetchGetURL**(), and **fetchPutURL**() constitute the recommended interface to the **fetch** library. They examine the URL passed to them to determine the transfer method, and call the appropriate lower-level functions to perform the actual transfer. **fetchXGetURL**() also returns the remote document's metadata in the *url_stat* structure pointed to by the *us* argument.

The *flags* argument is a string of characters which specify transfer options. The meaning of the individual flags is scheme-dependent, and is detailed in the appropriate section below.

fetchStatURL() attempts to obtain the requested document's metadata and fill in the structure pointed to by its second argument. The *url_stat* structure is defined as follows in <fetch.h>:

```
struct url_stat {
    off_t       size;
    time_t      atime;
    time_t      mtime;
};
```

If the size could not be obtained from the server, the *size* field is set to -1. If the modification time could not be obtained from the server, the *mtime* field is set to the epoch. If the access time could not be obtained from the server, the *atime* field is set to the modification time.

fetchListURL() attempts to list the contents of the directory pointed to by the URL provided. If successful, it returns a malloced array of *url_ent* structures. The *url_ent* structure is defined as follows in <fetch.h>:

```
struct url_ent {
    char         name[MAXPATHLEN];
    struct url_stat stat;
};
```

The list is terminated by an entry with an empty name.

The pointer returned by **fetchListURL**() should be freed using **free**().

fetchXGet(), **fetchGet**(), **fetchPut**() and **fetchStat**() are similar to **fetchXGetURL**(), **fetchGetURL**(), **fetchPutURL**() and **fetchStatURL**(), except that they expect a pre-parsed URL in the form of a pointer to a *struct url* rather than a string.

All of the **fetchXGetXXX**(), **fetchGetXXX**() and **fetchPutXXX**() functions return a pointer to a stream which can be used to read or write data from or to the requested document, respectively. Note that although the implementation details of the individual access methods vary, it can generally be assumed that a stream returned by one of the **fetchXGetXXX**() or **fetchGetXXX**() functions is read-only, and that a stream returned by one of the **fetchPutXXX**() functions is write-only.

FILE SCHEME

fetchXGetFile(), **fetchGetFile**() and **fetchPutFile**() provide access to documents which are files in a locally mounted file system. Only the <document> component of the URL is used.

fetchXGetFile() and **fetchGetFile**() do not accept any flags.

fetchPutFile() accepts the 'a' (append to file) flag. If that flag is specified, the data written to the stream returned by **fetchPutFile**() will be appended to the previous contents of the file, instead of replacing them.

FTP SCHEME

fetchXGetFTP(), **fetchGetFTP**() and **fetchPutFTP**() implement the FTP protocol as described in RFC959.

If the 'p' (passive) flag is specified, a passive (rather than active) connection will be attempted.

If the 'l' (low) flag is specified, data sockets will be allocated in the low (or default) port range instead of the high port range (see ip(4)).

If the 'd' (direct) flag is specified, **fetchXGetFTP**(), **fetchGetFTP**() and **fetchPutFTP**() will use a direct connection even if a proxy server is defined.

If no user name or password is given, the **fetch** library will attempt an anonymous login, with user name "anonymous" and password "anonymous@<hostname>".

HTTP SCHEME

The **fetchXGetHTTP**(), **fetchGetHTTP**() and **fetchPutHTTP**() functions implement the HTTP/1.1 protocol. With a little luck, there is even a chance that they comply with RFC2616 and RFC2617.

If the 'd' (direct) flag is specified, **fetchXGetHTTP**(), **fetchGetHTTP**() and **fetchPutHTTP**() will use a direct connection even if a proxy server is defined.

Since there seems to be no good way of implementing the HTTP PUT method in a manner consistent with the rest of the **fetch** library, **fetchPutHTTP**() is currently unimplemented.

AUTHENTICATION

Apart from setting the appropriate environment variables and specifying the user name and password in the URL or the *struct url*, the calling program has the option of defining an authentication function with the following prototype:

int **myAuthMethod**(*struct url *u*)

The callback function should fill in the *user* and *pwd* fields in the provided *struct url* and return 0 on success, or any other value to indicate failure.

To register the authentication callback, simply set *fetchAuthMethod* to point at it. The callback will be used whenever a site requires authentication and the appropriate environment variables are not set.

This interface is experimental and may be subject to change.

RETURN VALUES

fetchParseURL() returns a pointer to a *struct url* containing the individual components of the URL. If it is unable to allocate memory, or the URL is syntactically incorrect, **fetchParseURL**() returns a NULL pointer.

The **fetchStat**() functions return 0 on success and -1 on failure.

All other functions return a stream pointer which may be used to access the requested document, or NULL if an error occurred.

The following error codes are defined in <fetch.h>:

[FETCH_ABORT]	Operation aborted
[FETCH_AUTH]	Authentication failed
[FETCH_DOWN]	Service unavailable
[FETCH_EXISTS]	File exists
[FETCH_FULL]	File system full
[FETCH_INFO]	Informational response
[FETCH_MEMORY]	Insufficient memory
[FETCH_MOVED]	File has moved
[FETCH_NETWORK]	Network error
[FETCH_OK]	No error
[FETCH_PROTO]	Protocol error
[FETCH_RESOLV]	Resolver error
[FETCH_SERVER]	Server error
[FETCH_TEMP]	Temporary error
[FETCH_TIMEOUT]	Operation timed out
[FETCH_UNAVAIL]	File is not available
[FETCH_UNKNOWN]	Unknown error
[FETCH_URL]	Invalid URL

The accompanying error message includes a protocol-specific error code and message, e.g. "File is not available (404 Not Found)"

ENVIRONMENT

FETCH_BIND_ADDRESS	Specifies a hostname or IP address to which sockets used for outgoing connections will be bound.
FTP_LOGIN	Default FTP login if none was provided in the URL.
FTP_PASSIVE_MODE	If set to anything but 'no', forces the FTP code to use passive mode.
FTP_PASSWORD	Default FTP password if the remote server requests one and none was provided in the URL.
FTP_PROXY	URL of the proxy to use for FTP requests. The document part is ignored. FTP and HTTP proxies are supported; if no scheme is specified, FTP is assumed. If the proxy is an FTP proxy, **libfetch** will send user@host as user name to

the proxy, where `user` is the real user name, and `host` is the name of the FTP server.

If this variable is set to an empty string, no proxy will be used for FTP requests, even if the `HTTP_PROXY` variable is set.

`ftp_proxy` Same as `FTP_PROXY`, for compatibility.

`HTTP_AUTH` Specifies HTTP authorization parameters as a colon-separated list of items. The first and second item are the authorization scheme and realm respectively; further items are scheme-dependent. Currently, only basic authorization is supported.

Basic authorization requires two parameters: the user name and password, in that order.

This variable is only used if the server requires authorization and no user name or password was specified in the URL.

`HTTP_PROXY` URL of the proxy to use for HTTP requests. The document part is ignored. Only HTTP proxies are supported for HTTP requests. If no port number is specified, the default is 3128.

Note that this proxy will also be used for FTP documents, unless the `FTP_PROXY` variable is set.

`http_proxy` Same as `HTTP_PROXY`, for compatibility.

`HTTP_PROXY_AUTH` Specifies authorization parameters for the HTTP proxy in the same format as the `HTTP_AUTH` variable.

This variable is used if and only if connected to an HTTP proxy, and is ignored if a user and/or a password were specified in the proxy URL.

`HTTP_REFERER` Specifies the referrer URL to use for HTTP requests. If set to “auto”, the document URL will be used as referrer URL.

`HTTP_USER_AGENT` Specifies the User-Agent string to use for HTTP requests. This can be useful when working with HTTP origin or proxy servers that differentiate between user agents.

`NETRC` Specifies a file to use instead of `~/.netrc` to look up login names and passwords for FTP sites. See `ftp`(1) for a description of the file format. This feature is experimental.

EXAMPLES

To access a proxy server on `proxy.example.com` port 8080, set the `HTTP_PROXY` environment variable in a manner similar to this:

```
HTTP_PROXY=http://proxy.example.com:8080
```

If the proxy server requires authentication, there are two options available for passing the authentication data. The first method is by using the proxy URL:

```
HTTP_PROXY=http://<user>:<pwd>@proxy.example.com:8080
```

The second method is by using the `HTTP_PROXY_AUTH` environment variable:

```
HTTP_PROXY=http://proxy.example.com:8080
HTTP_PROXY_AUTH=basic:*:<user>:<pwd>
```

SEE ALSO

`fetch`(1), `ftpio`(3), `ip`(4)

J. Postel and J. K. Reynolds, *File Transfer Protocol*, October 1985, RFC959.

P. Deutsch, A. Emtage, and A. Marine., *How to Use Anonymous FTP*, May 1994, RFC1635.

T. Berners-Lee, L. Masinter, and M. McCahill, *Uniform Resource Locators (URL)*, December 1994, RFC1738.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol -- HTTP/1.1*, January 1999, RFC2616.

J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, *HTTP Authentication: Basic and Digest Access Authentication*, June 1999, RFC2617.

HISTORY

The **`fetch`** library first appeared in FreeBSD 3.0.

AUTHORS

The **`fetch`** library was mostly written by Dag-Erling Smørgrav ⟨des@FreeBSD.org⟩ with numerous suggestions from Jordan K. Hubbard ⟨jkh@FreeBSD.org⟩, Eugene Skepner ⟨eu@qub.com⟩ and other FreeBSD developers. It replaces the older **`ftpio`** library written by Poul-Henning Kamp ⟨phk@FreeBSD.org⟩ and Jordan K. Hubbard ⟨jkh@FreeBSD.org⟩.

This manual page was written by Dag-Erling Smørgrav ⟨des@FreeBSD.org⟩.

BUGS

Some parts of the library are not yet implemented. The most notable examples of this are **`fetchPutHTTP`**(), **`fetchListHTTP`**(), **`fetchListFTP`**() and FTP proxy support.

There is no way to select a proxy at run-time other than setting the `HTTP_PROXY` or `FTP_PROXY` environment variables as appropriate.

`libfetch` does not understand or obey 305 (Use Proxy) replies.

Error numbers are unique only within a certain context; the error codes used for FTP and HTTP overlap, as do those used for resolver and system errors. For instance, error code 202 means "Command not implemented, superfluous at this site" in an FTP context and "Accepted" in an HTTP context.

`fetchStatFTP`() does not check that the result of an MDTM command is a valid date.

The man page is incomplete, poorly written and produces badly formatted text.

The error reporting mechanism is unsatisfactory.

Some parts of the code are not fully reentrant.

NAME

fflush, **fpurge** — flush a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

int
fflush(FILE *stream);

int
fpurge(FILE *stream);
```

DESCRIPTION

The function **fflush**() forces a write of all buffered data for the given output or update *stream* via the stream's underlying write function. The open status of the stream is unaffected.

If the *stream* argument is NULL, **fflush**() flushes *all* open output streams.

The function **fpurge**() erases any input or output buffered in the given *stream*. For output streams this discards any unwritten output. For input streams this discards any input read from the underlying object but not yet obtained via getc(3); this includes any text pushed back via ungetc(3).

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, EOF is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EBADF] *stream* is not an open stream, or, in the case of **fflush**(), not a stream open for writing.

The function **fflush**() may also fail and set *errno* for any of the errors specified for the routine write(2).

SEE ALSO

write(2), fclose(3), fopen(3), setbuf(3)

STANDARDS

The **fflush**() function conforms to ANSI X3.159-1989 ("ANSI C89").

NAME

ffs — find first bit set in a bit string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <strings.h>

int
ffs(int value);
```

DESCRIPTION

The **ffs**() function finds the first bit set in *value* and returns the index of that bit. Bits are numbered starting from 1, starting at the right-most bit. A return value of 0 means that the argument was zero.

SEE ALSO

bitstring(3)

HISTORY

The **ffs**() function appeared in 4.3BSD.

NAME

fgetln — get a line from a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

char *
fgetln(FILE * restrict stream, size_t * restrict len);
```

DESCRIPTION

The **fgetln**() function returns a pointer to the next line from the stream referenced by *stream*. This line is *not* a C string as it does not end with a terminating NUL character. The length of the line, including the final newline, is stored in the memory location to which *len* points. (Note, however, that if the line is the last in a file that does not end in a newline, the returned text will not contain a newline.)

RETURN VALUES

Upon successful completion a pointer is returned; this pointer becomes invalid after the next I/O operation on *stream* (whether successful or not) or as soon as the stream is closed. Otherwise, NULL is returned. The **fgetln**() function does not distinguish between end-of-file and error; the routines feof(3) and ferror(3) must be used to determine which occurred. If an error occurs, the global variable *errno* is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return NULL until the condition is cleared with clearerr(3).

The text to which the returned pointer points may be modified, provided that no changes are made beyond the returned size. These changes are lost as soon as the pointer becomes invalid.

ERRORS

[EBADF] The argument *stream* is not a stream open for reading.

The **fgetln**() function may also fail and set *errno* for any of the errors specified for the routines fflush(3), malloc(3), read(2), stat(2), or realloc(3).

SEE ALSO

ferror(3), fgets(3), fopen(3), putc(3)

HISTORY

The **fgetln**() function first appeared in 4.4BSD.

CAVEATS

Since the returned buffer is not a C string (it is not null terminated), a common practice is to replace the newline character with '\0'. However, if the last line in a file does not contain a newline, the returned text won't contain a newline either. The following code demonstrates how to deal with this problem by allocating a temporary buffer:

```
char *buf, *lbuf;
size_t len;

lbuf = NULL;
while ((buf = fgetln(fp, &len))) {
        if (buf[len - 1] == '\n')
                buf[len - 1] = '\0';
```

```
                else {
                        if ((lbuf = (char *)malloc(len + 1)) == NULL)
                                err(1, NULL);
                        memcpy(lbuf, buf, len);
                        lbuf[len] = '\0';
                        buf = lbuf;
                }
                printf("%s\n", buf);

                if (lbuf != NULL) {
                        free(lbuf);
                        lbuf = NULL;
                }
        }
```

NAME

fgets, **gets** — get a line from a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

char *
fgets(char * restrict str, int size, FILE * restrict stream);

char *
gets(char *str);
```

DESCRIPTION

The **fgets**() function reads at most one less than the number of characters specified by *size* from the given *stream* and stores them in the string *str*. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained, and a '\0' character is appended to end the string.

The **gets**() function is equivalent to **fgets**() with an infinite *size* and a *stream* of *stdin*, except that the newline character (if any) is not stored in the string. It is the caller's responsibility to ensure that the input line, if any, is sufficiently short to fit in the string.

RETURN VALUES

Upon successful completion, **fgets**() and **gets**() return a pointer to the string. If end-of-file or an error occurs before any characters are read, they return NULL. The **fgets**() and **gets**() functions do not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to determine which occurred.

ERRORS

[EBADF] The given *stream* is not a readable stream.

The function **fgets**() may also fail and set *errno* for any of the errors specified for the routines fflush(3), fstat(2), read(2), or malloc(3).

The function **gets**() may also fail and set *errno* for any of the errors specified for the routine getchar(3).

SEE ALSO

feof(3), ferror(3), fgetln(3)

STANDARDS

The functions **fgets**() and **gets**() conform to ANSI X3.159-1989 ("ANSI C89").

CAVEATS

The following bit of code illustrates a case where the programmer assumes a string is too long if it does not contain a newline:

```
char buf[1024], *p;

while (fgets(buf, sizeof(buf), fp) != NULL) {
        if ((p = strchr(buf, '\n')) == NULL) {
                fprintf(stderr, "input line too long.\n");
                exit(1);
        }
```

```
                *p = '\0';
                printf("%s\n", buf);
        }
```

While the error would be true if a line > 1023 characters were read, it would be false in two other cases:

1. If the last line in a file does not contain a newline, the string returned by **fgets**() will not contain a newline either. Thus **strchr**() will return NULL and the program will terminate, even if the line was valid.

2. All C string functions, including **strchr**(), correctly assume the end of the string is represented by a null ('\0') character. If the first character of a line returned by **fgets**() were null, **strchr**() would immediately return without considering the rest of the returned text which may indeed include a newline.

Consider using fgetln(3) instead when dealing with untrusted input.

SECURITY CONSIDERATIONS

Since it is usually impossible to ensure that the next input line is less than some arbitrary length, and because overflowing the input buffer is almost invariably a security violation, programs should *NEVER* use **gets**(). The **gets**() function exists purely to conform to ANSI X3.159-1989 ("ANSI C89").

NAME

fgetwln — get a line of wide characters from a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wchar_t *
fgetwln(FILE * restrict stream, size_t * restrict len);
```

DESCRIPTION

The **fgetwln**() function returns a pointer to the next line from the stream referenced by *stream*. This line is *not* a standard wide character string as it does not end with a terminating null wide character. The length of the line, including the final newline, is stored in the memory location to which *len* points. (Note, however, that if the line is the last in a file that does not end in a newline, the returned text will not contain a newline.)

RETURN VALUES

Upon successful completion a pointer is returned; this pointer becomes invalid after the next I/O operation on *stream* (whether successful or not) or as soon as the stream is closed. Otherwise, NULL is returned. The **fgetwln**() function does not distinguish between end-of-file and error; the routines feof(3) and ferror(3) must be used to determine which occurred. If an error occurs, the global variable *errno* is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return NULL until the condition is cleared with clearerr(3).

The text to which the returned pointer points may be modified, provided that no changes are made beyond the returned size. These changes are lost as soon as the pointer becomes invalid.

ERRORS

[EBADF] The argument *stream* is not a stream open for reading.

The **fgetwln**() function may also fail and set *errno* for any of the errors specified for the routines mbrtowc(3), realloc(3), or read(2).

SEE ALSO

ferror(3), fgetln(3), fgetws(3), fopen(3)

NAME

fgetws — get a line of wide characters from a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wchar_t *
fgetws(wchar_t * restrict ws, int n, FILE * restrict fp);
```

DESCRIPTION

The **fgetws**() function reads at most one less than the number of characters specified by *n* from the given *fp* and stores them in the wide character string *ws*. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a '\0' character is appended to end the string.

RETURN VALUES

Upon successful completion, **fgetws**() returns *ws*. If end-of-file occurs before any characters are read, **fgetws**() returns NULL and the buffer contents remain unchanged. If an error occurs, **fgetws**() returns NULL and the buffer contents are indeterminate. The **fgetws**() function does not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to determine which occurred.

ERRORS

[EBADF] The given *fp* argument is not a readable stream.

[EILSEQ] The data obtained from the input stream does not form a valid multibyte character.

The function **fgetws**() may also fail and set *errno* for any of the errors specified for the routines fflush(3), fstat(2), read(2), or malloc(3).

SEE ALSO

feof(3), ferror(3), fgets(3)

STANDARDS

The **fgetws**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

flockfile, **ftrylockfile**, **funlockfile** — stdio stream locking functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

void
flockfile(FILE *file);

int
ftrylockfile(FILE *file);

void
funlockfile(FILE *file);
```

DESCRIPTION

The **flockfile**(), **ftrylockfile**(), and **funlockfile**() functions provide applications with explicit control of locking of stdio stream objects. They can be used by a thread to execute a sequence of I/O operations as a unit, without interference from another thread.

Locks on stdio streams are recursive, and a lock count is maintained. stdio streams are created unlocked, with a lock count of zero. After successful acquisition of the lock, its count is incremented to one, indicating locked state of the stdio stream. Each subsequent relock operation performed by the owner thread increments the lock count by one, and each subsequent unlock operation performed by the owner thread decrements the lock count by one, allowing matching lock and unlock operations to be nested. After its lock count is decremented to zero, the stdio stream returns to unlocked state, and ownership of the stdio stream is relinquished.

The **flockfile**() function acquires the ownership of *file* for the calling thread. If *file* is already owned by another thread, the calling thread is suspended until the acquisition is possible (i.e., *file* is relinquished again and the calling thread is scheduled to acquire it).

The **ftrylockfile**() function acquires the ownership of *file* for the calling thread only if *file* is available.

The **funlockfile**() function relinquishes the ownership of *file* previously granted to the calling thread. Only the current owner of *file* may **funlockfile**() it.

RETURN VALUES

If successful, the **ftrylockfile**() function returns 0. Otherwise, it returns non-zero to indicate that the lock cannot be acquired.

SEE ALSO

getc_unlocked(3), getchar_unlocked(3), putc_unlocked(3), putchar_unlocked(3)

STANDARDS

The **flockfile**(), **ftrylockfile**() and **funlockfile**() functions conform to IEEE Std 1003.1-2001 (“POSIX.1”).

BUGS

The design of these interfaces does not allow for addressing the problem of priority inversion.

NAME

floor, **floorf** — round to largest integral value not greater than x

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
floor(double x);

float
floorf(float x);
```

DESCRIPTION

The **floor**() and **floorf**() functions return the largest integral value less than or equal to *x*.

SEE ALSO

abs(3), ceil(3), fabs(3), ieee(3), math(3), rint(3)

STANDARDS

The **floor**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

fmod, **fmodf** — floating-point remainder function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
fmod(double x, double y);

float
fmodf(float x, float y);
```

DESCRIPTION

The **fmod**() function computes the floating-point remainder of *x*/*y*.

RETURN VALUES

The **fmod**() and **fmodf**() functions return the value *x*-*i*∗*y*, for some integer *i* such that, if *y* is non-zero, the result has the same sign as *x* and magnitude less than the magnitude of *y*. If *y* is zero, whether a domain error occurs or the **fmod**() function returns zero is implementation-defined.

SEE ALSO

math(3)

STANDARDS

The **fmod**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

fmtcheck — sanitizes user-supplied printf(3)-style format string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

const char *
fmtcheck(const char *fmt_suspect, const char *fmt_default);
```

DESCRIPTION

The **fmtcheck** function scans *fmt_suspect* and *fmt_default* to determine if *fmt_suspect* will consume the same argument types as *fmt_default* and to ensure that *fmt_suspect* is a valid format string.

The printf(3) family of functions can not verify the types of arguments that they are passed at run-time. In some cases, like catgets(3), it is useful or necessary to use a user-supplied format string with no guarantee that the format string matches the specified parameters.

The **fmtcheck** function was designed to be used in these cases, as in:

```
printf(fmtcheck(user_format, standard_format), arg1, arg2);
```

In the check, field widths, fillers, precisions, etc. are ignored (unless the field width or precision is an asterisk '*' instead of a digit string). Also, any text other than the format specifiers is completely ignored.

Note that the formats may be quite different as long as they accept the same parameters. For example, "%p %o %30s %#llx %-10.*e %n" is compatible with "This number %lu %d%% and string %s has %qd numbers and %.*g floats (%n)." However, "%o" is not equivalent to "%lx" because the first requires an integer and the second requires a long.

RETURN VALUES

If *fmt_suspect* is a valid format and consumes the same argument types as *fmt_default*, then the **fmtcheck** function will return *fmt_suspect*. Otherwise, it will return *fmt_default*.

SEE ALSO

printf(3)

NAME

fmtmsg — format and display a message

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <fmtmsg.h>

int
fmtmsg(long classification, const char *label, int severity,
    const char *text, const char *action, const char *tag);
```

DESCRIPTION

The **fmtmsg**() function can be used to display messages in the specified format. Messages may be written either to standard error, to the console, or both.

A formatted message consists of up to five components specified in *label*, *severity*, *text*, *action* and *tag*. Further information such as the origin of the message, the recoverability from the condition causing the message and where to display the message is specified in *classification*.

classification

The *classification* argument consists of a major classification and several sub-classifications. It has no effect on the content of the message displayed. With the exception of the display sub-classification, only a single identifier may be specified for each (sub-)classification. The following classifications are available:

Major Classifications	The source of the condition. Available identifiers are: MM_HARD (hardware), MM_SOFT (software), and MM_FIRM (firmware).
Message Source Sub-classifications	The type of software detecting the condition. Available identifiers are: MM_APPL (application), MM_UTIL (utility), and MM_OPSYS (operating system).
Display Sub-classifications	The displays the formatted messages is to be written to. Available identifiers are: MM_PRINT (standard error stream) and MM_CONSOLE (system console).
Status Sub-classifications	The capability of the calling software to recover from the condition. Available identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-recoverable).

If no *classification* is to be supplied, MM_NULLMC must be specified.

label

The *label* argument identifies the source of the message. It consists of two fields separated by a colon (:). The first field is up to 10 characters, the second is up to 14 characters.

If no *label* is to be supplied, MM_NULLLBL must be specified.

severity

The seriousness of the condition causing the message. The following *severity* levels are available:

MM_HALT The software has encountered a severe fault and is halting.

MM_ERROR The software has encountered a fault.

MM_WARNING The software has encountered an unusual non-fault condition.

MM_INFO The software informs about a non-error condition.

If no *severity* level is to be supplied, MM_NOSEV must be specified.

text

The description of the condition the software encountered. The character string is not limited to a specific size.

If no *text* is to be supplied, MM_NOTXT must be specified.

action

The first step to be taken to recover from the condition the software encountered; it will be preceded by the prefix "TO FIX:". The character string is not limited to a specific size.

If no *action* is to be supplied, MM_NOACT must be specified.

tag

The on-line documentation which provides further information about the condition and the message, such as "fmtmsg(3)". The character string is not limited to a specific size.

If no *tag* is to be supplied, MM_NOTAG must be specified.

Further effect on the formatting of the message as displayed on the standard error stream (but not on the system console!) may be taken by setting the MSGVERB environment variable, which selects the subset of message components to be printed. It consists of a colon-separated list of the optional keywords *label*, *severity*, *text*, *action*, and *tag*, which correspond to the arguments to **fmtmsg**() with the same names. If MSGVERB is either not set or malformed (containing empty or unknown keywords), its content is ignored an all message components will be selected.

Note that displaying a message on the system console may fail due to inappropriate privileges or a non-permissive file mode of the console device.

RETURN VALUES

The **fmtmsg**() function returns one of the following values:

MM_OK The function succeeded.

MM_NOTOK The function failed completely.

MM_NOMSG The function was unable to generate a message on standard error, but otherwise succeeded.

MM_NOCOM The function was unable to generate a message on the console, but otherwise succeeded.

SEE ALSO

printf(3), syslog(3)

STANDARDS

The **fmtmsg**() function conforms to X/Open System Interfaces and Headers Issue 5 ("XSH5").

NAME

fnmatch — match filename or pathname using shell glob rules

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <fnmatch.h>

int
fnmatch(const char *pattern, const char *string, int flags);
```

DESCRIPTION

The **fnmatch**() function matches patterns according to the globbing rules used by the shell. It checks the string specified by the *string* argument to see if it matches the pattern specified by the *pattern* argument.

The *flags* argument modifies the interpretation of *pattern* and *string*. The value of *flags* is the bitwise inclusive OR of any of the following constants, which are defined in the include file `fnmatch.h`.

`FNM_NOESCAPE`	Normally, every occurrence of a backslash (‘\’) followed by a character in *pattern* is replaced by that character. This is done to negate any special meaning for the character. If the `FNM_NOESCAPE` flag is set, a backslash character is treated as an ordinary character.
`FNM_PATHNAME`	Slash characters in *string* must be explicitly matched by slashes in *pattern*. If this flag is not set, then slashes are treated as regular characters.
`FNM_PERIOD`	Leading periods in strings match periods in patterns. The definition of ‘‘leading’’ is related to the specification of `FNM_PATHNAME`. A period is always ‘‘leading’’ if it is the first character in *string*. Additionally, if `FNM_PATHNAME` is set, a period is ‘‘leading’’ if it immediately follows a slash.
`FNM_LEADING_DIR`	Ignore “/*” rest after successful *pattern* matching.
`FNM_CASEFOLD`	The pattern is matched in a case-insensitive fashion.

RETURN VALUES

The **fnmatch**() function returns zero if *string* matches the pattern specified by *pattern*, otherwise, it returns the value `FNM_NOMATCH`.

SEE ALSO

`sh`(1), `glob`(3), `regex`(3)

STANDARDS

The **fnmatch**() function conforms to IEEE Std 1003.2-1992 (“POSIX.2”). The `FNM_CASEFOLD` flag is a NetBSD extension.

HISTORY

The **fnmatch**() function first appeared in 4.4BSD.

BUGS

The pattern ‘*’ matches the empty string, even if `FNM_PATHNAME` is specified.

NAME

fopen, **fdopen**, **freopen** — stream open functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

FILE *
fopen(const char * restrict path, const char * restrict mode);

FILE *
fdopen(int fildes, const char *mode);

FILE *
freopen(const char * restrict path, const char * restrict mode,
    FILE * restrict stream);
```

DESCRIPTION

The **fopen**() function opens the file whose name is the string pointed to by *path* and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

"r" Open for reading.

"r+" Open for reading and writing.

"w" Open for writing. Truncate file to zero length or create file.

"w+" Open for reading and writing. Truncate file to zero length or create file.

"a" Append; open for writing. The file is created if it does not exist.

"a+" Append; open for reading and writing. The file is created if it does not exist.

The *mode* string can also include the letter ''b'' either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with ANSI X3.159-1989 ("ANSI C89") and has no effect; the ''b'' is ignored.

The letter ''f'' in the mode string restricts fopen to regular files; if the file opened is not a regular file, **fopen**() will fail. This is a non ANSI X3.159-1989 ("ANSI C89") extension.

Any created files will have mode "S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH" (0666), as modified by the process' umask value (see umask(2)).

Opening a file with append mode causes all subsequent writes to it to be forced to the then current end of file, regardless of intervening repositioning of the stream.

The **fopen**() and **freopen**() functions initially position the stream at the start of the file unless the file is opened with append mode, in which case the stream is initially positioned at the end of the file.

The **fdopen**() function associates a stream with the existing file descriptor, *fildes*. The *mode* of the stream must be compatible with the mode of the file descriptor. The stream is positioned at the file offset of the file descriptor.

The **freopen**() function opens the file whose name is the string pointed to by *path* and associates the stream pointed to by *stream* with it. The original stream (if it exists) is closed. The *mode* argument is

used just as in the **fopen**() function. The primary use of the **freopen**() function is to change the file associated with a standard text stream (*stderr*, *stdin*, or *stdout*).

RETURN VALUES

Upon successful completion **fopen**(), **fdopen**() and **freopen**() return a FILE pointer. Otherwise, NULL is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] The *mode* provided to **fopen**(), **fdopen**(), or **freopen**() was invalid.

[EFTYPE] The file is not a regular file and the character ''f'' is specified in the mode.

The **fopen**(), **fdopen**() and **freopen**() functions may also fail and set *errno* for any of the errors specified for the routine malloc(3).

The **fopen**() function may also fail and set *errno* for any of the errors specified for the routine open(2).

The **fdopen**() function may also fail and set *errno* for any of the errors specified for the routine fcntl(2).

The **freopen**() function may also fail and set *errno* for any of the errors specified for the routines open(2), fclose(3) and fflush(3).

SEE ALSO

open(2), fclose(3), fileno(3), fseek(3), funopen(3)

STANDARDS

The **fopen**() and **freopen**() functions conform to ANSI X3.159-1989 (“ANSI C89”). The **fdopen**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

CAVEATS

Proper code using **fdopen**() with error checking should close(2) *fildes* in case of failure, and fclose(3) the resulting FILE * in case of success.

```
FILE *file;
int fd;

if ((file = fdopen(fd, "r")) != NULL) {
        /* perform operations on the FILE * */
        fclose(file);
} else {
        /* failure, report the error */
        close(fd);
}
```

NAME

pos_form_cursor — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

int
pos_form_cursor(FORM *form);
```

DESCRIPTION

The function **pos_form_cursor**() positions the screen cursor at the correct position for the form. This function can be used to restore the cursor state after using other curses routines.

RETURN VALUES

pos_form_cursor() will return one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	A bad argument was passed to the function.
E_NOT_POSTED	The form is not posted to the screen.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

form — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

int
data_ahead(FORM *form);

int
data_behind(FORM *form);
```

DESCRIPTION

If there is data offscreen to the right of the current field of the given form then **data_ahead**() will return TRUE, otherwise FALSE is returned. Similarly, if there is data offscreen to the left of the current field of the given form then **data_behind**() will return TRUE.

RETURN VALUES

If the condition is met then the functions will return TRUE, if there is an error or there is no data offscreen the functions will return FALSE.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

form_driver — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

int
form_driver(FORM *form, int request);
```

DESCRIPTION

The **form_driver**() is the heart of the forms library, it takes commands in the *request* parameter that is either a request to the driver to perform some action or is a character to be inserted into the current field. The form driver will attempt to insert any printable character passed to it into the current field. This may or may not succeed depending on the state of the current field. If the character passed is not printable then the driver attempts to process it as a driver request. If the character passed is not a valid request then the driver will return an unknown command error.

PARAMETERS

The forms driver recognizes the following requests:

REQ_NEXT_PAGE	Change to the next page in the form.
REQ_PREV_PAGE	Change to the previous page in the form.
REQ_FIRST_PAGE	Select the first page in the form.
REQ_LAST_PAGE	Go to the last page in the form.
REQ_NEXT_FIELD	Move to the next field in the form field array.
REQ_PREV_FIELD	Move to the previous field in the form field array.
REQ_FIRST_FIELD	Go to the first field in the form field array.
REQ_LAST_FIELD	Go to the last field in the form field array.
REQ_SNEXT_FIELD	Move to the next sorted field on the form.
REQ_SPREV_FIELD	Move to the previous sorted field on the form.
REQ_SFIRST_FIELD	Go to the first field in the sorted list.
REQ_SLAST_FIELD	Move to the last field in the sorted list.
REQ_LEFT_FIELD	Go one field to the left on the form page.
REQ_RIGHT_FIELD	Go one field to the right on the form page.
REQ_UP_FIELD	Go up one field on the form page.
REQ_DOWN_FIELD	Go down one field on the form page.
REQ_NEXT_CHAR	Move one char to the right within the field
REQ_PREV_CHAR	Move one char to the left within the current field.
REQ_NEXT_LINE	Go down one line in the current field.
REQ_PREV_LINE	Go up one line in the current field.
REQ_NEXT_WORD	Go forward one word in the current field
REQ_PREV_WORD	Go backward one word in the current field.
REQ_BEG_FIELD	Move the cursor to the beginning of the current field.
REQ_END_FIELD	Move the cursor to the end of the current field.
REQ_BEG_LINE	Move the cursor to the beginning of the line in the current field.
REQ_END_LINE	Move the cursor to the end of the line.
REQ_LEFT_CHAR	Move the cursor left one character

REQ_RIGHT_CHAR	Move the cursor right one character
REQ_UP_CHAR	Move the cursor up one line.
REQ_DOWN_CHAR	Move the cursor down one line.
REQ_NEW_LINE	Insert a new line at the current cursor position.
REQ_INS_CHAR	Insert a blank character at the current cursor position
REQ_INS_LINE	Open a blank line at the current cursor position.
REQ_DEL_CHAR	Delete the character at the current cursor position.
REQ_DEL_PREV	Delete the character to the left of the current cursor position.
REQ_DEL_LINE	Delete the current line.
REQ_DEL_WORD	Delete the word at the current cursor position.
REQ_CLR_EOL	Clear the field from the current cursor position to the end of the current line.
REQ_CLR_EOF	Clear the field from the current cursor position to the end of the field.
REQ_CLR_FIELD	Clear the field.
REQ_OVL_MODE	Enter overlay mode, characters added to the field will replace the ones already there.
REQ_INS_MODE	Enter insert mode, characters will be inserted at the current cursor position. Any characters to the right of the cursor will be moved right to accommodate the new characters.
REQ_SCR_FLINE	Scroll the field forward one line.
REQ_SCR_BLINE	Scroll the field backward one line.
REQ_SCR_FPAGE	Scroll the field forward one field page.
REQ_SCR_BPAGE	Scroll the field backward one field page.
REQ_SCR_FHPAGE	Scroll the field forward half one field page.
REQ_SCR_BHPAGE	Scroll the field backward half one field page.
REQ_SCR_FCHAR	Scroll the field horizontally forward one character
REQ_SCR_BCHAR	Scroll the field horizontally backward one character
REQ_SCR_HFLINE	Scroll the field horizontally forward one field line.
REQ_SCR_HBLINE	Scroll the field horizontally backward one field line.
REQ_SCR_HFHALF	Scroll the field horizontally forward half a field line.
REQ_SCR_HBHALF	Scroll the field horizontally backward half a field line.
REQ_VALIDATION	Request the contents of the current field be validated using any field validation function that has been set for the field. Normally, the field is validated before the current field changes. This request allows the current field to be validated.
REQ_PREV_CHOICE	Select the previous choice in an enumerated type field.
REQ_NEXT_CHOICE	Select the next choice in an enumerated type field.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following error values:

`E_OK`	The function was successful.
`E_REQUEST_DENIED`	The forms driver request could not be fulfilled
`E_UNKNOWN_COMMAND`	The passed character is not a printable character and is not a valid forms driver request.
`E_BAD_ARGUMENT`	A bad argument was passed to the forms driver.
`E_INVALID_FIELD`	The form passed to the driver has no valid attached fields.
`E_NOT_POSTED`	The given form is not currently posted to the screen.
`E_BAD_STATE`	The forms driver was called from within an init or term function.
`E_INVALID_FIELD`	The character passed to the forms driver fails the character validation for the current field.

SEE ALSO

`curses`(3), `forms`(3)

NOTES

Field sorting is done by location of the field on the form page, the fields are sorted by position starting with the top-most, left-most field and progressing left to right. For the purposes of sorting, the fields top left corner is used as the sort criteria. The header ⟨`form.h`⟩ automatically includes both ⟨`curses.h`⟩ and ⟨`eti.h`⟩.

NAME

field_count, **form_fields**, **move_field**, **set_form_fields** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

int
field_count(FORM *form);

FIELD **
form_fields(FORM *form);

int
move_field(FIELD *field, int frow, int fcol);

int
set_form_fields(FORM *form, FIELD **fields);
```

DESCRIPTION

The **field_count**() function returns the number of fields that are attached to the given form, if the form argument passed is NULL then **field_count**() will return −1. The function **form_fields**() will return a pointer to array of attach fields for the given form, this array is not NULL terminated, fields may be attached to the given form by calling **set_form_fields**(). The *fields* argument in this function is a pointer to a NULL terminated array of fields that will be attached to the form. If there are already fields attached to the form then they will be detached before the new fields are attached. The new fields given must not be attached to any other form. The **move_field**() function will move the given field to the location specified by *frow* and *fcol*.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following error values:

E_OK	The function was successful.
E_CONNECTED	The field is connected to a form.
E_POSTED	The form is currently posted to the screen.
E_BAD_ARGUMENT	The function was passed a bad argument.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

field_back, **field_fore**, **field_pad**, **set_field_back**, **set_field_fore**, **set_field_pad** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

chtype
field_back(FIELD *field);

chtype
field_fore(FIELD *field);

int
field_pad(FIELD *field);

int
set_field_back(FIELD *field, chtype attribute);

int
set_field_fore(FIELD *field, chtype attribute);

int
set_field_pad(FIELD *field, int pad);
```

DESCRIPTION

Calling the function **field_back**() will return the character attributes that will be applied to a field that is not the current field, these attributes can be set by the **set_field_back**() function. The **field_fore**() function returns the character attributes that will be used to indicate that a field is the currently active one on the form, this attribute may be set by using the **set_field_fore**() function. The pad character for a field is the character that will be printed in all field locations not occupied with actual field contents. The pad character can be retrieved by calling the **field_pad**() function, the pad character is set by using the **set_field_pad**() function.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following error values:

`E_OK`	The function was successful.

SEE ALSO

`curses`(3), `forms`(3)

NOTES

The header ⟨`form.h`⟩ automatically includes both ⟨`curses.h`⟩ and ⟨`eti.h`⟩.

NAME

field_buffer, **field_status**, **set_field_buffer**, **set_field_printf**, **set_field_status**, **set_max_field** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

char *
field_buffer(FIELD *field, int buffer);

int
field_status(FIELD *field);

int
set_field_buffer(FIELD *field, int buffer, char *value);

int
set_field_printf(FIELD *field, int buffer, char *fmt, ...);

int
set_field_status(FIELD *field, int status);

int
set_max_field(FIELD *field, int max);
```

DESCRIPTION

The **field_buffer**() function returns the contents of the buffer number specified by *buffer* for the given field. If the requested buffer number exceeds the number of buffers attached to the field then NULL will be returned. If the field option O_REFORMAT is enabled on the given field then storage will be allocated to hold the reformatted buffer. This storage must be release by calling free(3) when it is no longer required. If the O_REFORMAT field option is not set then no extra storage is allocated. The field buffer may be set by calling **set_field_buffer**() which will set the given buffer number to the contents of the string passed. A buffer may also be set by calling **set_field_printf**() which sets the buffer using the format arg *fmt* after being expanded using the subsequent arguments in the same manner as sprintf(3) does. Calling **field_status**() will return the status of the first buffer attached to the field. If the field has been modified then the function will return TRUE otherwise FALSE is returned, the status of the first buffer may be programmatically set by calling **set_field_status**(). The maximum growth of a dynamic field can be set by calling **set_max_field**() which limits the fields rows if the field is a multiline field or the fields columns if the field only has a single row.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	A bad parameter was passed to the function.
E_SYSTEM_ERROR	A system error occurred performing the function.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨`form.h`⟩ automatically includes both ⟨`curses.h`⟩ and ⟨`eti.h`⟩. The function **`set_field_printf`**() is a NetBSD extension and must not be used in portable code.

NAME

dynamic_field_info, **field_info** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

int
dynamic_field_info(FIELD *field, int *drows, int *dcols, int *max);

int
field_info(FIELD *field, int *rows, int *cols, int *frow, int *fcol,
    int *nrow, int *nbuf);
```

DESCRIPTION

The function **dynamic_field_info**() returns the sizing information for the field given. The function will return the number of rows, columns and the maximum growth of the field in the storage pointed to by the drows, dcols and max parameters respectively. Dynamic field information cannot be requested for the default field. If the field given is not dynamic then **dynamic_field_info**() will simply return the size of the actual field. The **field_info**() will return the number or rows, columns, field starting row, field starting column, number of off screen rows and number of buffers in *rows*, *cols*, *frow*, *fcol*, *nrow* and *nbuf* respectively.

RETURN VALUES

The functions will return one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	A bad argument was passed to the function.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

field_just, **set_field_just** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

int
field_just(FIELD *field);

int
set_field_just(FIELD *field, int justification);
```

DESCRIPTION

Field justification is only applied to static fields, a dynamic field will not be justified. The default justification for a field is NO_JUSTIFICATION. The **field_just**() will return the current justification value of the given field and the justification may be set by calling the **set_field_just**() function.

PARAMETERS

The following are the valid justifications for a field:

NO_JUSTIFICATION	No justification is to be applied to the field. In practice, this is the same as JUSTIFY_LEFT.
JUSTIFY_RIGHT	The field will be right justified. That is, the end of each line will be butted up against the right hand side of the field.
JUSTIFY_LEFT	The field will be left justified. That is, the start of each line will be butted up against the left hand side of the field.
JUSTIFY_CENTER	The field will be centre justified, padding will be applied to either end of the line to make the line centred in the field.

RETURN VALUES

The functions will return one of the following error values:

E_OK	The function was successful.
E_CURRENT	The field specified is the currently active one on the form.
E_BAD_ARGUMENT	A bad argument was passed to the function.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

dup_field, **free_field**, **link_field**, **new_field** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

FIELD *
dup_field(FIELD *field, int frow, int fcol);

int
free_field(FIELD *field);

FIELD *
link_field(FIELD *field, int frow, int fcol);

FIELD *
new_field(int rows,int cols,int frow,int fcol,int nrows,int nbuf);
```

DESCRIPTION

The **dup_field**() function duplicates the given field, including any buffers associated with the field and returns the pointer to the newly created field. **free_field**() destroys the field and frees any allocated resources associated with the field. The function **link_field**() copies the given field to a new field at the location *frow* and *fcol* but shares the buffers with the original field. **new_field**() creates a new field of size *rows* by *cols* at location *frow*, *fcol* on the page, the argument *nrows* specified the number of off screen rows the field has and the *nbuf* parameter specifies the number of extra buffers attached to the field. There will always be one buffer associated with a field.

RETURN VALUES

On error **dup_field**() and **new_field**() will return NULL. The functions will one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	A bad argument was passed to the function.
E_CONNECTED	The field is connected to a form.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

field_opts, **field_opts_off**, **field_opts_on**, **set_field_opts** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

Form_Options
field_opts(FIELD *field);

int
field_opts_off(FIELD *field, Form_Options options);

int
field_opts_on(FIELD *field, Form_Options options);

int
set_field_opts(FIELD *field, Form_Options options);
```

DESCRIPTION

The function **field_opts**() returns the current options settings for the given field. The **field_opts_off**() will turn the options given in *options* off for the given field, options not specified in *options* will remain unchanged. Conversely, the function **field_opts_on**() will turn on the options given in *options* for the specified field, again, any options not specified will remain unchanged. The options for a field may be set to a specific set of options by calling the **set_field_opts**() function. Options may only be changed if the field given is not the currently active one.

PARAMETERS

The following options are available for a field:

O_VISIBLE	The field is visible, hence is displayed when the form is posted.
O_ACTIVE	The field is active in the form, meaning that it can be visited during form processing.
O_PUBLIC	The contents of the field are echoed to the screen.
O_EDIT	The contents of the field can be modified
O_WRAP	The contents of the field are wrapped on a word boundary, if this option is off then the field will be wrapped on a character boundary.
O_BLANK	Blank the field on new data being entered if and only if the field cursor is at the left hand side of the field.
O_AUTOSKIP	Skip to the next field when the current field reaches its maximum size.
O_NULLOK	The field is allowed to contain no data
O_STATIC	The field is not dynamic, it has a fixed size.
O_PASSOK	An unmodified field is allowed.
O_REFORMAT	Retain the formatting of a field when the buffer is retrieved. If this option is not set then the buffer returned will be a single string with no line breaks. When this option is set newline characters will be inserted at the point where the string has been wrapped in a multiline field. This option is an extension to the forms library and must not be used in portable code. See the field_buffer(3) man page for how this option modifies the behaviour of **field_buffer**().

The following options are on by default for a field: O_VISIBLE, O_ACTIVE, O_PUBLIC, O_EDIT, O_WRAP, O_BLANK, O_AUTOSKIP, O_NULLOK, O_PASSOK, and O_STATIC.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following error values:

`E_OK`	The function was successful.
`E_CURRENT`	The field specified is the currently active one in the form.

SEE ALSO

`curses`(3), `forms`(3)

NOTES

The header ⟨`form.h`⟩ automatically includes both ⟨`curses.h`⟩ and ⟨`eti.h`⟩. The option `O_REFORMAT` is a NetBSD
extension and must not be used in portable code.

NAME

field_userptr, **set_field_userptr** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

void *
field_userptr(FIELD *field);

int
set_field_userptr(FIELD *field, void *ptr);
```

DESCRIPTION

The **field_userptr**() function returns the pointer to the user defined data for the field, this pointer may be set by calling the **set_field_userptr**() function.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following error values:

E_OK	The function was successful.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

field_arg, **field_type**, **set_field_type** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

char *
field_arg(FIELD *field);

FIELDTYPE *
field_type(FIELD *field);

int
set_field_type(FIELD *field, FIELDTYPE *type, ...);
```

DESCRIPTION

The **field_arg**() function returns the field type arguments that are associated with the given field. The **field_type**() function returns the field type structure associated with the given field, this type can be set by calling the **set_field_type**() function which associates the given field type with the field, the third and subsequent parameters are field dependent arguments.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following error values:

E_OK	The function was successful.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

free_fieldtype, **link_fieldtype**, **new_fieldtype**, **set_fieldtype_arg**, **set_fieldtype_choice** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

int
free_fieldtype(FIELDTYPE *fieldtype);

FIELDTYPE *
link_fieldtype(FIELDTYPE *type1, FIELDTYPE *type2);

FIELDTYPE *
new_fieldtype(int (*field_check)(FIELD *, char *),
    int (*char_check)(int, char *));

int
set_fieldtype_arg(FIELDTYPE *fieldtype,char * (*make_args)(va_list *),
    char * (*copy_args)(char *),void (*free_args)(char *));

int
set_fieldtype_choice(FIELDTYPE *fieldtype,
    int (*next_choice)(FIELD *, char *),
    int (*prev_choice)(FIELD *, char *));
```

DESCRIPTION

The function **free_fieldtype**() frees the storage associated with the field type and destroys it. The function **link_fieldtype**() links together the two given field types to produce a new field type. A new field type can be created by calling **new_fieldtype**() which requires pointers to two functions which perform validation, the *field_check* function must validate the field contents and return TRUE if they are acceptable and FALSE if they are not. The *char_check* validates the character input into the field, this function will be called for each character entered, if the character can be entered into the field then *char_check* must return TRUE. Neither *field_check* nor *char_check* may be NULL. The functions for handling the field type arguments can be defined by using the **set_fieldtype_arg**() function, the *make_args* function is used to create new arguments for the fieldtype, the *copy_args* is used to copy the fieldtype arguments to a new arguments structure and *free_args* is used to destroy the fieldtype arguments and release any associated storage, none of these function pointers may be NULL. The field type choice functions can be set by calling **set_fieldtype_choice**(), the *next_choice* and *prev_choice* specify the next and previous choice functions for the field type. These functions must perform the necessary actions to select the next or previous choice for the field, updating the field buffer if necessary. The choice functions must return TRUE if the function succeeded and FALSE otherwise.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	The function was passed a bad argument.

`E_CONNECTED` The field is connected to a form.

SEE ALSO

`curses`(3), `forms`(3)

NOTES

The header ⟨`form.h`⟩ automatically includes both ⟨`curses.h`⟩ and ⟨`eti.h`⟩.

NAME

field_init, **field_term**, **form_init**, **form_term**, **set_field_init**, **set_field_term**, **set_form_init**, **set_form_term** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

void (*)(FORM *)
field_init(FORM *form);

void (*)(FORM *)
field_term(FORM *form);

void (*)(FORM *)
form_init(FORM *form);

void (*)(FORM *)
form_term(FORM *form);

int
set_field_init(FORM *form, void (*function)(FORM *));

int
set_field_term(FORM *form, void (*function)(FORM *));

int
set_form_init(FORM *form, void (*function)(FORM *));

int
set_form_term(FORM *form, void (*function)(FORM *));
```

DESCRIPTION

The **field_init**() function returns a pointer to the function that will be called just after the current field changes and just before the form is posted, this function may be set by using the **set_field_init**() function. Similarly, the function **field_term**() will return a pointer to the function that will be called just before the current field changes and just after the form is unposted, this function pointer may be set by using the **set_field_term**() function. The **form_init**() function will return a pointer to the function that will be called just before the form is posted to the screen, this function can be set by calling the **set_form_init**() function. The **form_term**() function will return a pointer to the function that will be called just after the form is unposted from the screen, this function may be set by using the **set_form_term**() function. By default, the init and term function pointers are NULL.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following error values:

E_OK	The function was successful.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨`form.h`⟩ automatically includes both ⟨`curses.h`⟩ and ⟨`eti.h`⟩.

NAME

free_form, **new_form** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

int
free_form(FORM *form);

FORM *
new_form(FIELD **fields);
```

DESCRIPTION

The function **free_form**() frees all the resources associated with the form and destroys the form. Calling **new_form**() will create a new form, set the form parameters to the current defaults and attach the passed fields to the form. The array of fields passed to **new_form**() must be terminated with a NULL pointer to indicate the end of the fields.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	The function was passed a bad argument.
E_POSTED	The form is posted to the screen.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

new_page, **set_new_page** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

int
new_page(FIELD *field);

int
set_new_page(FIELD *field, int page);
```

DESCRIPTION

The **new_page**() function returns TRUE if the given field is the start of a new page, otherwise it returns FALSE, the new page status of a field can be set or unset using the **set_new_page**() function.

RETURN VALUES

The functions will return one of the following error values:

E_OK	The function was successful.
E_CONNECTED	The field is connected to a form.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

form_opts, **form_opts_off**, **form_opts_on**, **set_form_opts** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

Form_Options
form_opts(FORM *form);

int
form_opts_off(FORM *form, Form_Options options);

int
form_opts_on(FORM *form, Form_Options options);

int
set_form_opts(FORM *form, Form_Options options);
```

DESCRIPTION

The function **form_opts**() returns the current options that are set on the given form. The **form_opts_off**() will turn off the form options given in *options* for the form, similarly, **form_opts_on**() will turn on the options specified in *options* for the given form. The form options can be set to an explicit set by calling **set_form_opts**().

PARAMETERS

The following form options are valid:

O_BS_OVERLOAD
: If this option is set and the cursor is at the first character in the field then the backspace character will perform the same function as a REQ_PREV_FIELD driver request, moving to the previous field in the form.

O_NL_OVERLOAD
: If this option is set and the cursor is at the end of the field then the new line character will perform the same function as a REQ_NEXT_FIELD driver request, moving to the next field in the form.

By default no form options are set.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following error values:

`E_OK`	The function was successful.

SEE ALSO

`curses`(3), `forms`(3)

NOTES

The header ⟨`form.h`⟩ automatically includes both ⟨`curses.h`⟩ and ⟨`eti.h`⟩.

NAME

current_field, **field_index**, **form_page**, **form_max_page**, **set_current_field**, **set_form_page** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

FIELD *
current_field(FORM *form);

int
field_index(FIELD *field);

int
form_page(FORM *form);

int
form_max_page(FORM *form);

int
set_current_field(FORM *form, FIELD *field);

int
set_form_page(FORM *form, int page);
```

DESCRIPTION

The **current_field**() returns a pointer to the structure for the field that is currently active on the page. If there is an error, **current_field**() will return NULL. Calling **field_index**() will return the index of the given field in the form field array. The current page the form is on can be determined by using **form_page**(), the current page of a form can be programmatically set by calling **set_form_page**(). The maximum page number for a form can be found by calling the function **form_max_page**() but note that this function is a NetBSD extension and must not be used in portable forms library programs. The current field on the form may be set by calling **set_current_field**() which will set the current field to the one given.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	The function was passed a bad argument.
E_NOT_CONNECTED	The given field is not associated with a form.
E_BAD_STATE	The function was called from within an init or term function.
E_INVALID_FIELD	The field given is not part of the given form.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

The **form_max_page** is a NetBSD extension and should not be used in portable applications.

NAME

post_form, **unpost_form** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

int
post_form(FORM *form);

int
unpost_form(FORM *form);
```

DESCRIPTION

The **post_form**() function performs the actions necessary to present the form on the curses screen. If there are any init functions that need to be called then they will be called prior to the form being posted and the cursor will be positioned on the first active field that can be visited. Conversely, the function **unpost_form**() removes the form from the screen and calls any termination functions that were specified.

RETURN VALUES

The functions will return one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	A bad argument was passed to the function.
E_POSTED	The form is already posted to the screen.
E_NOT_POSTED	The form was not posted to the screen.
E_NOT_CONNECTED	There are no fields associated with the form.
E_BAD_STATE	The function was called from within a init or term function.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

form_userptr, **set_form_userptr** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

void *
form_userptr(FORM *form);

int
set_form_userptr(FORM *form, void *ptr);
```

DESCRIPTION

The **form_userptr**() function returns the pointer to the user defined data associated with the form, this pointer may be set using the **set_form_userptr**() call.

RETURN VALUES

Functions returning pointers will return `NULL` if an error is detected. The functions that return an int will return one of the following error values:

`E_OK`	The function was successful.

SEE ALSO

`curses`(3), `forms`(3)

NOTES

The header ⟨`form.h`⟩ automatically includes both ⟨`curses.h`⟩ and ⟨`eti.h`⟩.

NAME

form_sub, **form_win**, **scale_form**, **set_form_sub**, **set_form_win** — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>

WINDOW *
form_sub(FORM *form);

WINDOW *
form_win(FORM *form);

int
scale_form(FORM *form, int *rows, int *cols);

int
set_form_sub(FORM *form, WINDOW *window);

int
set_form_win(FORM *form, WINDOW *window);
```

DESCRIPTION

All output to the screen done by the forms library is handled by the curses library routines. By default, the forms library will output to the curses *stdscr*, but if the forms window has been set via **set_form_win**() then output will be sent to the window specified by **set_form_win**(), unless the forms subwindow has been set using **set_form_sub**(). If a subwindow has been specified using **set_form_sub**() then it will be used by the forms library to for screen output. The current setting for the form window can be retrieved by calling **form_win**(). If the forms window has not been set then NULL will be returned. Similarly, the forms subwindow can be found by calling the **form_sub**() function, again, if the subwindow has not been set then NULL will be returned. The **scale_form**() function will return the minimum number of rows and columns that will entirely contain the given form.

RETURN VALUES

Functions returning pointers will return NULL if an error is detected. The functions that return an int will return one of the following error values:

E_OK	The function was successful.
E_NOT_CONNECTED	The form has no fields connected to it.
E_POSTED	The form is posted to the screen.

SEE ALSO

curses(3), forms(3)

NOTES

The header ⟨form.h⟩ automatically includes both ⟨curses.h⟩ and ⟨eti.h⟩.

NAME

form — form library

LIBRARY

Curses Form Library (libform, −lform)

SYNOPSIS

```
#include <form.h>
```

DESCRIPTION

The **form** library provides a terminal independent form system using the curses(3) library. Before using the **form** functions the terminal must be set up by curses(3) using the **initscr**() function or similar. Programs using **form** functions must be linked with the curses(3) library **−lcurses**.

The **form** library provides facilities for defining form fields, placing a form on the terminal screen, assign pre and post change operations and setting the attributes of both the form and its fields.

Defining default attributes for forms and fields

The **form** library allows any settable attribute or option of both the form and field objects to be defined such that any new form or field automatically inherits the value as default. Setting the default value will not affect any field or form that has already been created but will be applied to subsequent objects. To set the default attribute or option the set routine is passed a NULL pointer in the field or form parameter when calling the set routine. The current default value can be retrieved by calling the get routine with a NULL pointer for the field or form parameter.

Form Routine Name	Manual Page Name
current_field	form_page(3)
data_ahead	form_data(3)
data_behind	form_data(3)
dup_field	form_field_new(3)
dynamic_field_info	form_field_info(3)
field_arg	form_field_validation(3)
field_back	form_field_attributes(3)
field_buffer	form_field_buffer(3)
field_count	form_field(3)
field_fore	form_field_attributes(3)
field_index	form_page(3)
field_info	form_field_info(3)
field_init	form_hook(3)
field_just	form_field_just(3)
field_opts	form_field_opts(3)
field_opts_off	form_field_opts(3)
field_opts_on	form_field_opts(3)
field_pad	form_field_attributes(3)
field_status	form_field_buffer(3)
field_term	form_hook(3)
field_type	form_field_validation(3)
field_userptr	form_field_userptr(3)
form_driver	form_driver(3)
form_fields	form_field(3)

form_init	form_hook(3)
form_max_page	form_page(3)
form_opts	form_opts(3)
form_opts_off	form_opts(3)
form_opts_on	form_opts(3)
form_page	form_page(3)
form_sub	form_win(3)
form_term	form_hook(3)
form_userptr	form_userptr(3)
form_win	form_win(3)
free_field	form_field_new(3)
free_fieldtype	form_fieldtype(3)
free_form	form_new(3)
link_field	form_field_new(3)
link_fieldtype	form_fieldtype(3)
move_field	form_field(3)
new_field	form_field_new(3)
new_fieldtype	form_fieldtype(3)
new_form	form_new(3)
new_page	form_new_page(3)
pos_form_cursor	form_cursor(3)
post_form	form_post(3)
scale_form	form_win(3)
set_current_field	form_page(3)
set_field_back	form_field_attributes(3)
set_field_buffer	form_field_buffer(3)
set_field_fore	form_field_attributes(3)
set_field_init	form_hook(3)
set_field_just	form_field_just(3)
set_field_opts	form_field_opts(3)
set_field_pad	form_field_attributes(3)
set_field_printf	form_field_buffer(3)
set_field_status	form_field_buffer(3)
set_field_term	form_hook(3)
set_field_type	form_field_validation(3)
set_field_userptr	form_field_userptr(3)
set_fieldtype_arg	form_fieldtype(3)
set_fieldtype_choice	form_fieldtype(3)
set_form_fields	form_field(3)
set_form_init	form_hook(3)
set_form_opts	form_opts(3)
set_form_page	form_page(3)
set_form_sub	form_win(3)
set_form_term	form_hook(3)
set_form_userptr	form_userptr(3)
set_form_win	form_win(3)
set_max_field	form_field_buffer(3)
set_new_page	form_new_page(3)
unpost_form	form_post(3)

RETURN VALUES

Any function returning a string pointer will return `NULL` if an error occurs. Functions returning an integer will return one of the following:

`E_OK`	The function was successful.
`E_SYSTEM_ERROR`	There was a system error during the call.
`E_BAD_ARGUMENT`	One or more of the arguments passed to the function was incorrect.
`E_POSTED`	The form is already posted.
`E_CONNECTED`	A field was already connected to a form.
`E_BAD_STATE`	The function was called from within an initialization or termination routine.
`E_NO_ROOM`	The form does not fit within the subwindow.
`E_NOT_POSTED`	The form is not posted.
`E_UNKNOWN_COMMAND`	The form driver does not recognize the request passed to it.
`E_NOT_SELECTABLE`	The field could not be selected.
`E_NOT_CONNECTED`	The field is not connected to a form.
`E_REQUEST_DENIED`	The form driver could not process the request.
`E_INVALID_FIELD`	The field is invalid.
`E_CURRENT`	The field is the active one on the form.

SEE ALSO

`curses`(3), `menus`(3)

NOTES

This implementation of the forms library does depart in behavior subtly from the original AT&T implementation. Some of the more notable departures are:

field wrapping	For multi-line fields the data will be wrapped as it is entered, this does not happen in the AT&T implementation.
buffer 0	In this implementation, the contents of buffer 0 are always current regardless of whether the field has been validated or not.
circular fields	In the AT&T implementation fields are circular on a page, that is, a next field from the last field will go to the first field on the current page. In this implementation a next field request on the last field of a page will result in the forms library positioning the cursor on the first field of the next page. If the field is the last field in the form then going to the next field will be denied, in the AT&T it would result in the cursor being placed on the first field of the first page.
buffer returns	In this implementation only the data entered by the user in the form field will be returned, unlike the AT&T library which would return the contents of the field padded to the size of the field with the pad character.
The TAB character	The handling of the TAB character in fields varies between implementations. In ncurses attempting to set a field contents with a string containing a TAB will result in an error and will not allow a TAB to be entered into a field. The AT&T library statically converts tabs to the equivalent number of spaces when the field buffer is set but the form driver will not allow a TAB to be inserted into the field buffer. This implementation allows TAB when setting the field buffer and also will allow TAB to be inserted into a field buffer via the form driver and correctly calculates the cursor position allowing for expansion of the TAB character.
set_field_printf	This function is a NetBSD extension and must not be used in portable code.
`O_REFORMAT`	This field option is a NetBSD extension and must not be used in portable code.

NAME

fparseln — return the next logical line from a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

char *
fparseln(FILE *stream, size_t *len, size_t *lineno, const char delim[3],
    int flags);
```

DESCRIPTION

The **fparseln**() function returns a pointer to the next logical line from the stream referenced by *stream*. This string is NUL terminated and it is dynamically allocated on each invocation. It is the responsibility of the caller to free the pointer.

By default, if a character is escaped, both it and the preceding escape character will be present in the returned string. Various *flags* alter this behaviour.

The meaning of the arguments is as follows:

stream The stream to read from.

len If not NULL, the length of the string is stored in the memory location to which it points.

lineno If not NULL, the value of the memory location to which is pointed to, is incremented by the number of lines actually read from the file.

delim Contains the escape, continuation, and comment characters. If a character is NUL then processing for that character is disabled. If NULL, all characters default to values specified below. The contents of *delim* is as follows:

- *delim[0]* The escape character, which defaults to \, is used to remove any special meaning from the next character.
- *delim[1]* The continuation character, which defaults to \, is used to indicate that the next line should be concatenated with the current one if this character is the last character on the current line and is not escaped.
- *delim[2]* The comment character, which defaults to #, if not escaped indicates the beginning of a comment that extends until the end of the current line.

flags If non-zero, alter the operation of **fparseln**(). The various flags, which may be *or*-ed together, are:

- FPARSELN_UNESCCOMM Remove escape preceding an escaped comment.
- FPARSELN_UNESCCONT Remove escape preceding an escaped continuation.
- FPARSELN_UNESCESC Remove escape preceding an escaped escape.
- FPARSELN_UNESCREST Remove escape preceding any other character.
- FPARSELN_UNESCALL All of the above.

RETURN VALUES

Upon successful completion a pointer to the parsed line is returned; otherwise, NULL is returned.

The **fparseln**() function uses internally `fgetln`(3), so all error conditions that apply to `fgetln`(3), apply to **fparseln**(). In addition **fparseln**() may set *errno* to [`ENOMEM`] and return `NULL` if it runs out of memory.

SEE ALSO

`fgetln`(3)

HISTORY

The **fparseln**() function first appeared in NetBSD 1.4.

NAME

fpclassify — classify real floating type

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <math.h>

int
fpclassify(real-floating x);
```

DESCRIPTION

The **fpclassify**() macro performs classification of its argument *x*. An argument represented in a format wider than its semantic type is converted to its semantic type first. The classification is then based on the type of the argument.

IEEE 754

FP_INFINITE	infinity, either positive or negative
FP_NAN	not-a-number (“NaN”)
FP_NORMAL	normal
FP_SUBNORMAL	subnormal
FP_ZERO	zero

VAX

FP_ROP	reserved operand (“ROP”)
FP_DIRTYZERO	dirty zero
FP_NORMAL	finite
FP_ZERO	true zero

RETURN VALUES

The **fpclassify**() macro returns the value of the number classification macro appropriate to its argument *x* as described above.

ERRORS

No errors are defined.

SEE ALSO

isfinite(3), isnormal(3), math(3), signbit(3)

STANDARDS

The **fpclassify**() macro conforms to ISO/IEC 9899:1999 (“ISO C99”).

NAME

fpgetmask, **fpgetround**, **fpgetsticky**, **fpsetmask**, **fpsetround**, **fpsetsticky** — IEEE FP mode control

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ieeefp.h>

fp_except
fpgetmask(void);

fp_rnd
fpgetround(void);

fp_except
fpgetsticky(void);

fp_except
fpsetmask(fp_except mask);

fp_rnd
fpsetround(fp_rnd rnd_dir);

fp_except
fpsetsticky(fp_except sticky);
```

DESCRIPTION

A rounding mode is one of FP_RZ, FP_RM, FP_RN, or FP_RP, for rounding towards zero, rounding (*Minus infinity*) down, rounding to *nearest*, and rounding (*Plus infinity*) up. The default mode is FP_RN.

An *fp_except* value is a bitmask specifying an exception type and containing any of the values listed below.

FP_X_INV	Invalid Operation
FP_X_DZ	Division by zero
FP_X_OFL	Overflow
FP_X_UFL	Underflow
FP_X_IMP	Imprecision (inexact)
FP_X_IOV	Integer Overflow

The **fpsetmask**() function will set the current exception mask, i.e., it will cause future operations with the specified result status to raise the SIGFPE exception. The **fpgetmask**() function will return the current exception mask.

The **fpsetround**() function will cause future operations to use the specified dynamic rounding mode. The **fpgetround**() function will return the current rounding mode.

Note: On some architectures, instructions can optionally specify static rounding modes and exception enables that will supersede the specified dynamic mode. On other architectures, these features may not be fully supported.

A “sticky” status word may be maintained in which a bit is set every time an exceptional floating point condition is encountered, whether or not a SIGFPE is generated. The **fpsetsticky**() function will set or clear the specified exception history bits. The **fpgetsticky**() function will return the exception history bits.

RETURN VALUES

The **fpgetround**() and **fpsetround**() functions return the (previous) rounding mode. The **fpgetmask**(), **fpsetmask**(), **fpgetsticky**(), and **fpsetsticky**() functions return the (previous) exception mask and exception history bits.

SEE ALSO

sigaction(2)

NAME

fputs, **puts** — output a line to a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

int
fputs(const char * restrict str, FILE * restrict stream);

int
puts(const char *str);
```

DESCRIPTION

The function **fputs**() writes the string pointed to by *str* to the stream pointed to by *stream*.

The function **puts**() writes the string *str*, and a terminating newline character, to the stream *stdout*.

RETURN VALUES

The **fputs**() function returns 0 on success and EOF on error; **puts**() returns a nonnegative integer on success and EOF on error.

ERRORS

[EBADF] The *stream* supplied is not a writable stream.

The functions **fputs**() and **puts**() may also fail and set *errno* for any of the errors specified for the routines write(2).

SEE ALSO

ferror(3), putc(3), stdio(3)

STANDARDS

The functions **fputs**() and **puts**() conform to ANSI X3.159-1989 (“ANSI C89”).

NAME

fputws — output a line of wide characters to a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int
fputws(const wchar_t * restrict ws, FILE * restrict fp);
```

DESCRIPTION

The **fputws**() function writes the wide character string pointed to by *ws* to the stream pointed to by *fp*.

RETURN VALUES

The **fputws**() function returns 0 on success and −1 on error.

ERRORS

[EBADF] The *fp* argument supplied is not a writable stream.

The **fputws**() function may also fail and set *errno* for any of the errors specified for the routine write(2).

SEE ALSO

ferror(3), fputs(3), putwc(3), stdio(3)

STANDARDS

The **fputws**() function conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

fread, **fwrite** — binary stream input/output

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

size_t
fread(void * restrict ptr, size_t size, size_t nmemb,
    FILE * restrict stream);

size_t
fwrite(const void * restrict ptr, size_t size, size_t nmemb,
    FILE * restrict stream);
```

DESCRIPTION

The function **fread**() reads *nmemb* objects, each *size* bytes long, from the stream pointed to by *stream*, storing them at the location given by *ptr*.

The function **fwrite**() writes *nmemb* objects, each *size* bytes long, to the stream pointed to by *stream*, obtaining them from the location given by *ptr*.

RETURN VALUES

The functions **fread**() and **fwrite**() advance the file position indicator for the stream by the number of bytes read or written. They return the number of objects read or written. If *size* or *nmemb* is 0, the functions return 0 and the state of *stream* remains unchanged. If an error occurs, or the end-of-file is reached, the return value is a short object count (or zero).

The function **fread**() does not distinguish between end-of-file and error, and callers must use `feof`(3) and `ferror`(3) to determine which occurred. The function **fwrite**() returns a value less than *nmemb* only if a write error has occurred.

SEE ALSO

`read`(2), `write`(2)

STANDARDS

The functions **fread**() and **fwrite**() conform to ANSI X3.159-1989 (“ANSI C89”).

NAME

frexp — convert floating-point number to fractional and integral components

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
frexp(double value, int *exp);

float
frexpf(float value, int *exp);
```

DESCRIPTION

The **frexp**() function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the *int* object pointed to by *exp*.

RETURN VALUES

The **frexp**() function returns the value *x*, such that *x* is a *double* with magnitude in the interval [1/2, 1) or zero, and *value* equals *x* times 2 raised to the power **exp*. If *value* is zero, both parts of the result are zero.

SEE ALSO

ldexp(3), math(3), modf(3)

STANDARDS

The **frexp**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

fgetpos, **fseek**, **fseeko**, **fsetpos**, **ftell**, **ftello**, **rewind** — reposition a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

int
fseek(FILE *stream, long int offset, int whence);

int
fseeko(FILE *stream, off_t offset, int whence);

long int
ftell(FILE *stream);

off_t
ftello(FILE *stream);

void
rewind(FILE *stream);

int
fgetpos(FILE * restrict stream, fpos_t * restrict pos);

int
fsetpos(FILE * restrict stream, const fpos_t * restrict pos);
```

DESCRIPTION

The **fseek**() function sets the file position indicator for the stream pointed to by *stream*. The new position, measured in bytes, is obtained by adding *offset* bytes to the position specified by *whence*. If *whence* is set to SEEK_SET, SEEK_CUR, or SEEK_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the **fseek**() function clears the end-of-file indicator for the stream and undoes any effects of the ungetc(3) function on the same stream.

The **fseeko**() function is identical to the **fseek**() function except that the *offset* argument is of type *off_t*.

The **ftell**() function obtains the current value of the file position indicator for the stream pointed to by *stream*.

The **ftello**() function is identical to the **ftell**() function except that the return value is of type *off_t*.

The **rewind**() function sets the file position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to:

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared (see clearerr(3)).

In this implementation, the **fgetpos**() and **fsetpos**() functions are alternative interfaces equivalent to **ftell**(), **ftello**(), **fseek**() and **fseeko**() (with whence set to SEEK_SET), setting and storing the current value of the file offset into or from the object referenced by *pos*. In others implementations, an "*fpos_t*" object may be a complex object and these routines may be the only way to portably reposition a text stream.

RETURN VALUES

The **rewind**() function returns no value. Upon successful completion, **fgetpos**(), **fseek**(), **fsetpos**() return 0, and **ftell**() returns the current offset. Otherwise, **fseek**() and **ftell**() return −1 and the others return a nonzero value and the global variable *errno* is set to indicate the error.

ERRORS

[EBADF] The *stream* specified is not a seekable stream.

[EINVAL] The *whence* argument to **fseek**() was not SEEK_SET, SEEK_END, or SEEK_CUR.

The function **fgetpos**(), **fseek**(), **fseeko**(), **fsetpos**(), **ftell**(), **ftello**(), and **rewind**() may also fail and set *errno* for any of the errors specified for the routines fflush(3), fstat(2), lseek(2), and malloc(3).

SEE ALSO

lseek(2)

STANDARDS

The **fgetpos**(), **fsetpos**(), **fseek**(), **ftell**(), and **rewind**() functions conform to ANSI X3.159-1989 ("ANSI C89"). The **fseeko**() and **ftello**() functions conform to X/Open System Interfaces and Headers Issue 5 ("XSH5").

BUGS

The **fgetpos**() and **fsetpos**() functions don't store/set shift states of the stream in this implementation.

NAME

ftime — get date and time

LIBRARY

Compatibility Library (libcompat, −lcompat)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/timeb.h>

int
ftime(struct timeb *tp);
```

DESCRIPTION

This interface is obsoleted by gettimeofday(2)**. It is available from the compatibility library, libcompat.**

Note: time zone information is no longer provided by this interface. See localtime(3) **for information on how to retrieve it.**

The **ftime**() routine fills in a structure pointed to by its argument, as defined by ⟨sys/timeb.h⟩:

```
struct timeb {
        time_t  time;
        unsigned short millitm;
        short   timezone;
        short   dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval. The *timezone* and *dstflag* fields are provided for source compatibility, but are always zero.

SEE ALSO

gettimeofday(2), settimeofday(2), ctime(3), localtime(3), time(3)

HISTORY

The **ftime**() function appeared in 4.1BSD.

NAME

ftok — create IPC identifier from path name

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t
ftok(const char *path, int id);
```

DESCRIPTION

The **ftok**() function attempts to create a unique key suitable for use with the msgget(2), semget(2) and shmget(2) functions given the *path* of an existing file and a user-selectable *id*.

The specified *path* must specify an existing file that is accessible to the calling process or the call will fail. Also, note that links to files will return the same key, given the same *id*.

Only the 8 least significant bits of *id* are used in the key generation; the rest of the bits are ignored.

RETURN VALUES

The **ftok**() function will return ((key_t)-1) if *path* does not exist or if it cannot be accessed by the calling process.

SEE ALSO

msgget(2), semget(2), shmget(2)

HISTORY

The **ftok**() function originated with System V and is typically used by programs that use the System V IPC routines.

AUTHORS

Thorsten Lockert <tholo@sigmasoft.com>

BUGS

The returned key is computed based on the device and inode of the specified *path* in combination with the given *id*. Thus it is quite possible for the routine to return duplicate keys given that those fields are not 8- and 16-bit quantities like they were on System V based systems where this library routine's ancestor were originally created.

NAME

fts, **fts_open**, **fts_read**, **fts_children**, **fts_set**, **fts_close** — traverse a file hierarchy

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fts.h>

FTS *
fts_open(char * const *path_argv, int options,
    int (*compar)(const FTSENT **, const FTSENT **));

FTSENT *
fts_read(FTS *ftsp);

FTSENT *
fts_children(FTS *ftsp, int options);

int
fts_set(FTS *ftsp, FTSENT *f, int options);

int
fts_close(FTS *ftsp);
```

DESCRIPTION

The **fts** functions are provided for traversing UNIX file hierarchies. A simple overview is that the **fts_open**() function returns a "handle" on a file hierarchy, which is then supplied to the other **fts** functions. The function **fts_read**() returns a pointer to a structure describing one of the files in the file hierarchy. The function **fts_children**() returns a pointer to a linked list of structures, each of which describes one of the files contained in a directory in the hierarchy. In general, directories are visited two distinguishable times; in pre-order (before any of their descendants are visited) and in post-order (after all of their descendants have been visited). Files are visited once. It is possible to walk the hierarchy "logically" (ignoring symbolic links) or physically (visiting symbolic links), order the walk of the hierarchy or prune and/or re-visit portions of the hierarchy.

Two structures are defined (and typedef'd) in the include file ⟨fts.h⟩. The first is *FTS*, the structure that represents the file hierarchy itself. The second is *FTSENT*, the structure that represents a file in the file hierarchy. Normally, an *FTSENT* structure is returned for every file in the file hierarchy. In this manual page, "file" and "*FTSENT* structure" are generally interchangeable. The *FTSENT* structure contains at least the following fields, which are described in greater detail below:

```
typedef struct _ftsent {
        u_short fts_info;           /* flags for FTSENT structure */
        char *fts_accpath;          /* access path */
        char *fts_path;             /* root path */
        short fts_pathlen;          /* strlen(fts_path) */
        char *fts_name;             /* file name */
        short fts_namelen;          /* strlen(fts_name) */
        short fts_level;            /* depth (-1 to N) */
        int fts_errno;              /* file errno */
        long fts_number;            /* local numeric value */
        void *fts_pointer;          /* local address value */
```

```
        struct ftsent *fts_parent;    /* parent directory */
        struct ftsent *fts_link;      /* next file structure */
        struct ftsent *fts_cycle;     /* cycle structure */
        struct stat *fts_statp;               /* stat(2) information */
} FTSENT;
```

These fields are defined as follows:

fts_info One of the following flags describing the returned *FTSENT* structure and the file it represents. With the exception of directories without errors (FTS_D), all of these entries are terminal, that is, they will not be revisited, nor will any of their descendants be visited.

- FTS_D — A directory being visited in pre-order.
- FTS_DC — A directory that causes a cycle in the tree. (The *fts_cycle* field of the *FTSENT* structure will be filled in as well).
- FTS_DEFAULT — Any *FTSENT* structure that represents a file type not explicitly described by one of the other *fts_info* values.
- FTS_DNR — A directory which cannot be read. This is an error return, and the *fts_errno* field will be set to indicate what caused the error.
- FTS_DOT — A file named '.' or '..' which was not specified as a file name to **fts_open**() (see FTS_SEEDOT).
- FTS_DP — A directory being visited in post-order. The contents of the *FTSENT* structure will be unchanged from when it was returned in pre-order, i.e., with the *fts_info* field set to FTS_D.
- FTS_ERR — This is an error return, and the *fts_errno* field will be set to indicate what caused the error.
- FTS_F — A regular file.
- FTS_NS — A file for which no stat(2) information was available. The contents of the *fts_statp* field are undefined. This is an error return, and the *fts_errno* field will be set to indicate what caused the error.
- FTS_NSOK — A file for which no stat(2) information was requested. The contents of the *fts_statp* field are undefined.
- FTS_SL — A symbolic link.
- FTS_SLNONE — A symbolic link with a non-existent target. The contents of the *fts_statp* field reference the file characteristic information for the symbolic link itself.
- FTS_W — A whiteout object.

fts_accpath A path for accessing the file from the current directory.

fts_path The path for the file relative to the root of the traversal. This path contains the path specified to **fts_open**() as a prefix.

fts_pathlen The length of the string referenced by *fts_path*.

fts_name The name of the file.

fts_namelen The length of the string referenced by *fts_name*.

fts_level The depth of the traversal, numbered from −1 to N, where this file was found. The *FTSENT* structure representing the parent of the starting point (or root) of the traversal is numbered −1, and the *FTSENT* structure for the root itself is numbered 0.

fts_errno Upon return of a *FTSENT* structure from the **fts_children**() or **fts_read**() functions, with its *fts_info* field set to FTS_DNR, FTS_ERR or FTS_NS, the *fts_errno* field contains the value of the external variable *errno* specifying the cause of the error. Otherwise, the contents of the *fts_errno* field are undefined.

fts_number This field is provided for the use of the application program and is not modified by the **fts** functions. It is initialized to 0.

fts_pointer This field is provided for the use of the application program and is not modified by the **fts** functions. It is initialized to NULL.

fts_parent A pointer to the *FTSENT* structure referencing the file in the hierarchy immediately above the current file, i.e., the directory of which this file is a member. A parent structure for the initial entry point is provided as well, however, only the *fts_level*, *fts_number* and *fts_pointer* fields are guaranteed to be initialized.

fts_link Upon return from the **fts_children**() function, the *fts_link* field points to the next structure in the NULL-terminated linked list of directory members. Otherwise, the contents of the *fts_link* field are undefined.

fts_cycle If a directory causes a cycle in the hierarchy (see FTS_DC), either because of a hard link between two directories, or a symbolic link pointing to a directory, the *fts_cycle* field of the structure will point to the *FTSENT* structure in the hierarchy that references the same file as the current *FTSENT* structure. Otherwise, the contents of the *fts_cycle* field are undefined.

fts_statp A pointer to stat(2) information for the file.

A single buffer is used for all of the paths of all of the files in the file hierarchy. Therefore, the *fts_path* and *fts_accpath* fields are guaranteed to be NULL-terminated *only* for the file most recently returned by **fts_read**(). To use these fields to reference any files represented by other *FTSENT* structures will require that the path buffer be modified using the information contained in that *FTSENT* structure's *fts_pathlen* field. Any such modifications should be undone before further calls to **fts_read**() are attempted. The *fts_name* field is always NULL-terminated.

FTS_OPEN

The **fts_open**() function takes a pointer to an array of character pointers naming one or more paths which make up a logical file hierarchy to be traversed. The array must be terminated by a NULL pointer.

There are a number of options, at least one of which (either FTS_LOGICAL or FTS_PHYSICAL) must be specified. The options are selected by *or*'ing the following values:

FTS_COMFOLLOW
This option causes any symbolic link specified as a root path to be followed immediately whether or not FTS_LOGICAL is also specified.

FTS_LOGICAL This option causes the **fts** routines to return *FTSENT* structures for the targets of symbolic links instead of the symbolic links themselves. If this option is set, the only symbolic links for which *FTSENT* structures are returned to the application are those referencing non-existent files. Either FTS_LOGICAL or FTS_PHYSICAL *must* be provided to the **fts_open**() function.

FTS_NOCHDIR As a performance optimization, the **fts** functions change directories as they walk the file hierarchy. This has the side-effect that an application cannot rely on being in any particular directory during the traversal. The FTS_NOCHDIR option turns off this optimization, and the **fts** functions will not change the current directory. Note that applications should not themselves change their current directory and try to access files unless FTS_NOCHDIR is specified and absolute pathnames were provided as arguments to **fts_open**().

FTS_NOSTAT By default, returned *FTSENT* structures reference file characteristic information (the *statp* field) for each file visited. This option relaxes that requirement as a performance optimization, allowing the **fts** functions to set the *fts_info* field to FTS_NSOK and leave the contents of the *statp* field undefined.

FTS_PHYSICAL This option causes the **fts** routines to return *FTSENT* structures for symbolic links themselves instead of the target files they point to. If this option is set, *FTSENT* structures for all symbolic links in the hierarchy are returned to the application. Either FTS_LOGICAL or FTS_PHYSICAL *must* be provided to the **fts_open**() function.

FTS_SEEDOT By default, unless they are specified as path arguments to **fts_open**(), any files named '.' or '..' encountered in the file hierarchy are ignored. This option causes the **fts** routines to return *FTSENT* structures for them.

FTS_WHITEOUT Return whiteout entries, which are normally hidden.

FTS_XDEV This option prevents **fts** from descending into directories that have a different device number than the file from which the descent began.

The argument **compar**() specifies a user-defined function which may be used to order the traversal of the hierarchy. It takes two pointers to pointers to *FTSENT* structures as arguments and should return a negative value, zero, or a positive value to indicate if the file referenced by its first argument comes before, in any order with respect to, or after, the file referenced by its second argument. The *fts_accpath*, *fts_path* and *fts_pathlen* fields of the *FTSENT* structures may *never* be used in this comparison. If the *fts_info* field is set to FTS_NS or FTS_NSOK, the *fts_statp* field may not either. If the **compar**() argument is NULL, the directory traversal order is in the order listed in *path_argv* for the root paths, and in the order listed in the directory for everything else.

FTS_READ

The **fts_read**() function returns a pointer to an *FTSENT* structure describing a file in the hierarchy. Directories (that are readable and do not cause cycles) are visited at least twice, once in pre-order and once in post-order. All other files are visited at least once. (Hard links between directories that do not cause cycles or symbolic links to symbolic links may cause files to be visited more than once, or directories more than twice.)

If all the members of the hierarchy have been returned, **fts_read**() returns NULL and sets the external variable *errno* to 0. If an error unrelated to a file in the hierarchy occurs, **fts_read**() returns NULL and sets *errno* appropriately. If an error related to a returned file occurs, a pointer to an *FTSENT* structure is returned, and *errno* may or may not have been set (see *fts_info*).

The *FTSENT* structures returned by **fts_read**() may be overwritten after a call to **fts_close**() on the same file hierarchy stream, or, after a call to **fts_read**() on the same file hierarchy stream unless they represent a file of type directory, in which case they will not be overwritten until after a call to **fts_read**() after the *FTSENT* structure has been returned by the function **fts_read**() in post-order.

FTS_CHILDREN

The **fts_children**() function returns a pointer to an *FTSENT* structure describing the first entry in a NULL-terminated linked list of the files in the directory represented by the *FTSENT* structure most recently returned by **fts_read**(). The list is linked through the *fts_link* field of the *FTSENT* structure, and is ordered by the user-specified comparison function, if any. Repeated calls to **fts_children**() will recreate this linked list.

As a special case, if **fts_read**() has not yet been called for a hierarchy, **fts_children**() will return a pointer to the files in the logical directory specified to **fts_open**(), i.e., the arguments specified to **fts_open**(). Otherwise, if the *FTSENT* structure most recently returned by **fts_read**() is not a directory being visited in pre-order, or the directory does not contain any files, **fts_children**() returns NULL and sets *errno* to zero. If an error occurs, **fts_children**() returns NULL and sets *errno* appropriately.

The *FTSENT* structures returned by **fts_children**() may be overwritten after a call to **fts_children**(), **fts_close**() or **fts_read**() on the same file hierarchy stream.

Option may be set to the following value:

FTS_NAMEONLY Only the names of the files are needed. The contents of all the fields in the returned linked list of structures are undefined with the exception of the *fts_name* and *fts_namelen* fields.

FTS_SET

The function **fts_set**() allows the user application to determine further processing for the file *f* of the stream *ftsp*. The **fts_set**() function returns 0 on success, and −1 if an error occurs. *Option* must be set to one of the following values:

FTS_AGAIN Re-visit the file; any file type may be re-visited. The next call to **fts_read**() will return the referenced file. The *fts_stat* and *fts_info* fields of the structure will be reinitialized at that time, but no other fields will have been changed. This option is meaningful only for the most recently returned file from **fts_read**(). Normal use is for post-order directory visits, where it causes the directory to be re-visited (in both pre and post-order) as well as all of its descendants.

FTS_FOLLOW The referenced file must be a symbolic link. If the referenced file is the one most recently returned by **fts_read**(), the next call to **fts_read**() returns the file with the *fts_info* and *fts_statp* fields reinitialized to reflect the target of the symbolic link instead of the symbolic link itself. If the file is one of those most recently returned by **fts_children**(), the *fts_info* and *fts_statp* fields of the structure, when returned by **fts_read**(), will reflect the target of the symbolic link instead of the symbolic link itself. In either case, if the target of the symbolic link does not exist the fields of the returned structure will be unchanged and the *fts_info* field will be set to FTS_SLNONE.

If the target of the link is a directory, the pre-order return, followed by the return of all of its descendants, followed by a post-order return, is done.

FTS_SKIP No descendants of this file are visited. The file may be one of those most recently returned by either **fts_children**() or **fts_read**().

FTS_CLOSE

The **fts_close**() function closes a file hierarchy stream *ftsp* and restores the current directory to the directory from which **fts_open**() was called to open *ftsp*. The **fts_close**() function returns 0 on success, and −1 if an error occurs.

ERRORS

The function **fts_open**() may fail and set *errno* for any of the errors specified for the library functions open(2) and malloc(3).

The function **fts_close**() may fail and set *errno* for any of the errors specified for the library functions chdir(2) and close(2).

The functions **fts_read**() and **fts_children**() may fail and set *errno* for any of the errors specified for the library functions chdir(2), malloc(3), opendir(3), readdir(3) and stat(2).

In addition, **fts_children**(), **fts_open**() and **fts_set**() may fail and set *errno* as follows:

[EINVAL] The options were invalid.

SEE ALSO

find(1), chdir(2), stat(2), qsort(3), symlink(7)

STANDARDS

The **fts** utility is expected to be included in a future IEEE Std 1003.1-1988 (“POSIX.1”) revision.

NAME

ftw, nftw — traverse (walk) a file tree

SYNOPSIS

```
#include <ftw.h>

int
ftw(const char *path, int (*fn)(const char *, const struct stat *, int),
    int maxfds);

int
nftw(const char *path,
    int (*fn)(const char *, const struct stat *, int, struct FTW *),
    int maxfds, int flags);
```

DESCRIPTION

These functions are provided for compatibility with legacy code. New code should use the `fts`(**3**) **functions.**

The **ftw**() and **nftw**() functions traverse (walk) the directory hierarchy rooted in *path*. For each object in the hierarchy, these functions call the function pointed to by *fn*. The **ftw**() function passes this function a pointer to a NUL-terminated string containing the name of the object, a pointer to a stat structure corresponding to the object, and an integer flag. The **nftw**() function passes the aforementioned arguments plus a pointer to a `FTW` structure as defined by ⟨`ftw.h`⟩ (shown below):

```
struct FTW {
    int base;  /* offset of basename into pathname */
    int level; /* directory depth relative to starting point */
};
```

Possible values for the flag passed to *fn* are:

`FTW_F` A regular file.

`FTW_D` A directory being visited in pre-order.

`FTW_DNR` A directory which cannot be read. The directory will not be descended into.

`FTW_DP` A directory being visited in post-order (**nftw**() only).

`FTW_NS` A file for which no `stat`(2) information was available. The contents of the stat structure are undefined.

`FTW_SL` A symbolic link.

`FTW_SLN` A symbolic link with a non-existent target (**nftw**() only).

The **ftw**() function traverses the tree in pre-order. That is, it processes the directory before the directory's contents.

The *maxfds* argument specifies the maximum number of file descriptors to keep open while traversing the tree. It has no effect in this implementation.

The **nftw**() function has an additional *flags* argument with the following possible values:

`FTW_PHYS` Physical walk, don't follow symbolic links.

`FTW_MOUNT` The walk will not cross a mount point.

FTW_DEPTH
: Process directories in post-order. Contents of a directory are visited before the directory itself. By default, **nftw**() traverses the tree in pre-order.

FTW_CHDIR
: Change to a directory before reading it. By default, **nftw**() will change its starting directory. The current working directory will be restored to its original value before **nftw**() returns.

RETURN VALUES

If the tree was traversed successfully, the **ftw**() and **nftw**() functions return 0. If the function pointed to by *fn* returns a non-zero value, **ftw**() and **nftw**() will stop processing the tree and return the value from *fn*. Both functions return −1 if an error is detected.

ERRORS

The **ftw**() and **nftw**() functions may fail and set *errno* for any of the errors specified for the library functions close(2), open(2), stat(2), malloc(3), opendir(3), and readdir(3). If the FGTW_CHDIR flag is set, the **nftw**() function may fail and set *errno* for any of the errors specified for chdir(2). In addition, either function may fail and set *errno* as follows:

[EINVAL] The *maxfds* argument is less than 1 or greater than OPEN_MAX.

SEE ALSO

chdir(2), close(2), open(2), stat(2), fts(3), malloc(3), opendir(3), readdir(3)

STANDARDS

The **ftw**() and **nftw**() functions conform to IEEE Std 1003.1-2001 (“POSIX.1”).

BUGS

The *maxfds* argument is currently ignored.

NAME

funopen, **fropen**, **fwopen** — open a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

FILE *
funopen(void *cookie, int (*readfn)(void *, char *, int),
    int (*writefn)(void *, const char *, int),
    fpos_t (*seekfn)(void *, fpos_t, int), int (*closefn)(void *));

FILE *
fropen(void *cookie, int (*readfn)(void *, char *, int));

FILE *
fwopen(void *cookie, int (*writefn)(void *, const char *, int));
```

DESCRIPTION

The **funopen**() function associates a stream with up to four "I/O functions". Either *readfn* or *writefn* must be specified; the others can be given as an appropriately-typed NULL pointer. These I/O functions will be used to read, write, seek and close the new stream.

In general, omitting a function means that any attempt to perform the associated operation on the resulting stream will fail. If the close function is omitted, closing the stream will flush any buffered output and then succeed.

The calling conventions of *readfn*, *writefn*, *seekfn* and *closefn* must match those, respectively, of read(2), write(2), lseek(2), and close(2); except that they are passed the *cookie* argument specified to **funopen**() in place of the traditional file descriptor argument, and *seekfn* uses fpos_t instead of off_t.

Read and write I/O functions are allowed to change the underlying buffer on fully buffered or line buffered streams by calling setvbuf(3). They are also not required to completely fill or empty the buffer. They are not, however, allowed to change streams from unbuffered to buffered or to change the state of the line buffering flag. They must also be prepared to have read or write calls occur on buffers other than the one most recently specified.

All user I/O functions can report an error by returning −1. Additionally, all of the functions should set the external variable *errno* appropriately if an error occurs.

An error on **closefn**() does not keep the stream open.

As a convenience, the include file ⟨stdio.h⟩ defines the macros **fropen**() and **fwopen**() as calls to **funopen**() with only a read or write function specified.

RETURN VALUES

Upon successful completion, **funopen**() returns a FILE pointer. Otherwise, NULL is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] The **funopen**() function was called without either a read or write function. The **funopen**() function may also fail and set *errno* for any of the errors specified for the routine malloc(3).

SEE ALSO

`fcntl`(2), `open`(2), `fclose`(3), `fopen`(3), `fseek`(3), `setbuf`(3)

HISTORY

The **funopen**() functions first appeared in 4.4BSD.

BUGS

The **funopen**() function may not be portable to systems other than BSD.

NAME

fwide — get/set orientation of a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int
fwide(FILE *stream, int mode);
```

DESCRIPTION

The **fwide**() function determines the orientation of the stream pointed at by *stream*.

If the orientation of *stream* has already been determined, **fwide**() leaves it unchanged. Otherwise, **fwide**() sets the orientation of *stream* according to *mode*.

If *mode* is less than zero, *stream* is set to byte-oriented. If it is greater than zero, *stream* is set to wide-oriented. Otherwise, *mode* is zero, and *stream* is unchanged.

RETURN VALUES

fwide() returns a value according to orientation after the call of **fwide**(); a value less than zero if byte-oriented, a value greater than zero if wide-oriented, and zero if the stream has no orientation.

SEE ALSO

ferror(3), fgetc(3), fgetwc(3), fopen(3), fputc(3), fputwc(3), freopen(3), stdio(3)

STANDARDS

The **fwide**() function conforms to ISO/IEC 9899:1999 (“ISO C99”).

NAME

gai_strerror — get error message string from EAI_xxx error code

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

const char *
gai_strerror(int ecode);
```

DESCRIPTION

The **gai_strerror**() function returns an error message string corresponding to the error code returned by getaddrinfo(3) or getnameinfo(3).

The following error codes and their meaning are defined in ⟨netdb.h⟩:

EAI_ADDRFAMILY	address family for *hostname* not supported
EAI_AGAIN	temporary failure in name resolution
EAI_BADFLAGS	invalid value for *ai_flags*
EAI_BADHINTS	invalid value for *hints*
EAI_FAIL	non-recoverable failure in name resolution
EAI_FAMILY	*ai_family* not supported.
EAI_MEMORY	memory allocation failure
EAI_NODATA	no address associated with *hostname*
EAI_NONAME	*hostname* or *servname* not provided, or not known
EAI_OVERFLOW	argument buffer overflow
EAI_PROTOCOL	resolved protocol is unknown
EAI_SERVICE	*servname* not supported for *ai_socktype*
EAI_SOCKTYPE	*ai_socktype* not supported
EAI_SYSTEM	system error returned in *errno*

RETURN VALUES

gai_strerror() returns a pointer to the error message string corresponding to *ecode*. If *ecode* is out of range, an implementation-specific error message string is returned.

SEE ALSO

getaddrinfo(3), getnameinfo(3)

NAME

GCQ_INIT, **GCQ_INIT_HEAD**, **gcq_init**, **gcq_init_head**, **gcq_q**, **gcq_hq**, **gcq_head**, **gcq_remove**, **gcq_onlist**, **gcq_empty**, **gcq_linked**, **gcq_insert_after**, **gcq_insert_before**, **gcq_insert_head**, **gcq_insert_tail**, **gcq_tie**, **gcq_tie_after**, **gcq_tie_before**, **gcq_merge**, **gcq_merge_head**, **gcq_merge_tail**, **gcq_clear**, **gcq_remove_all**, **GCQ_ITEM**, **GCQ_GOT_FIRST**, **GCQ_GOT_LAST**, **GCQ_GOT_NEXT**, **GCQ_GOT_PREV**, **GCQ_DEQUEUED_FIRST**, **GCQ_DEQUEUED_LAST**, **GCQ_DEQUEUED_NEXT**, **GCQ_DEQUEUED_PREV**, **GCQ_GOT_FIRST_TYPED**, **GCQ_GOT_LAST_TYPED**, **GCQ_GOT_NEXT_TYPED**, **GCQ_GOT_PREV_TYPED**, **GCQ_DEQUEUED_FIRST_TYPED**, **GCQ_DEQUEUED_LAST_TYPED**, **GCQ_DEQUEUED_NEXT_TYPED**, **GCQ_DEQUEUED_PREV_TYPED**, **GCQ_GOT_FIRST_COND**, **GCQ_GOT_LAST_COND**, **GCQ_GOT_NEXT_COND**, **GCQ_GOT_PREV_COND**, **GCQ_DEQUEUED_FIRST_COND**, **GCQ_DEQUEUED_LAST_COND**, **GCQ_DEQUEUED_NEXT_COND**, **GCQ_DEQUEUED_PREV_COND**, **GCQ_GOT_FIRST_COND_TYPED**, **GCQ_GOT_LAST_COND_TYPED**, **GCQ_GOT_NEXT_COND_TYPED**, **GCQ_GOT_PREV_COND_TYPED**, **GCQ_DEQUEUED_FIRST_COND_TYPED**, **GCQ_DEQUEUED_LAST_COND_TYPED**, **GCQ_DEQUEUED_NEXT_COND_TYPED**, **GCQ_DEQUEUED_PREV_COND_TYPED**, **GCQ_FOREACH**, **GCQ_FOREACH_REV**, **GCQ_FOREACH_NVAR**, **GCQ_FOREACH_NVAR_REV**, **GCQ_FOREACH_RO**, **GCQ_FOREACH_RO_REV**, **GCQ_FOREACH_DEQUEUED**, **GCQ_FOREACH_DEQUEUED_REV**, **GCQ_FOREACH_TYPED**, **GCQ_FOREACH_REV_TYPED**, **GCQ_FOREACH_NVAR_TYPED**, **GCQ_FOREACH_NVAR_REV_TYPED**, **GCQ_FOREACH_RO_TYPED**, **GCQ_FOREACH_RO_REV_TYPED**, **GCQ_FOREACH_DEQUEUED_TYPED**, **GCQ_FOREACH_DEQUEUED_REV_TYPED**, **GCQ_FIND**, **GCQ_FIND_REV**, **GCQ_FIND_TYPED**, **GCQ_FIND_REV_TYPED** — Generic Circular Queues

SYNOPSIS

```
#include <sys/gcq.h>

struct gcq;
struct gcq_head;

GCQ_INIT(name);

GCQ_INIT_HEAD(name);

static inline void
gcq_init(struct gcq *q);

static inline void
gcq_init_head(struct gcq_head *head);

static inline struct gcq *
gcq_q(struct gcq_head *head);

static inline struct gcq *
gcq_hq(struct gcq_head *head);

static inline struct gcq_head *
gcq_head(struct gcq *q);

static inline struct gcq *
gcq_remove(struct gcq *q);

static inline bool
gcq_onlist(struct gcq *q);

static inline bool
gcq_empty(struct gcq_head *head);
```

static inline bool
gcq_linked(*struct gcq *prev*, *struct gcq *next*);

static inline void
gcq_insert_after(*struct gcq *on*, *struct gcq *off*);

static inline void
gcq_insert_before(*struct gcq *on*, *struct gcq *off*);

static inline void
gcq_insert_head(*struct gcq_head *head*, *struct gcq *q*);

static inline void
gcq_insert_tail(*struct gcq_head *head*, *struct gcq *q*);

static inline void
gcq_tie(*struct gcq *dst*, *struct gcq *src*);

static inline void
gcq_tie_after(*struct gcq *dst*, *struct gcq *src*);

static inline void
gcq_tie_before(*struct gcq *dst*, *struct gcq *src*);

static inline void
gcq_merge(*struct gcq *dst*, *struct gcq *src*);

static inline void
gcq_merge_tail(*struct gcq_head *dst*, *struct gcq_head *src*);

static inline void
gcq_merge_head(*struct gcq_head *dst*, *struct gcq_head *src*);

static inline void
gcq_clear(*struct gcq *q*);

static inline void
gcq_remove_all(*struct gcq_head *head*);

*type **
GCQ_ITEM(*q*, *type*, *name*);

bool
GCQ_GOT_FIRST(*var*, *head*);

bool
GCQ_GOT_LAST(*var*, *head*);

bool
GCQ_GOT_NEXT(*var*, *current*, *head*, *start*);

bool
GCQ_GOT_PREV(*var*, *current*, *head*, *start*);

bool
GCQ_DEQUEUED_FIRST(*var*, *head*);

bool
GCQ_DEQUEUED_LAST(*var*, *head*);

bool
GCQ_DEQUEUED_NEXT(*var*, *current*, *head*, *start*);

bool
GCQ_DEQUEUED_PREV(*var*, *current*, *head*, *start*);

bool
GCQ_GOT_FIRST_TYPED(*tvar*, *head*, *type*, *name*);

bool
GCQ_GOT_LAST_TYPED(*tvar*, *head*, *type*, *name*);

bool
GCQ_GOT_NEXT_TYPED(*tvar*, *current*, *head*, *start*, *type*, *name*);

bool
GCQ_GOT_PREV_TYPED(*tvar*, *current*, *head*, *start*, *type*, *name*);

bool
GCQ_DEQUEUED_FIRST_TYPED(*tvar*, *head*, *type*, *name*);

bool
GCQ_DEQUEUED_LAST_TYPED(*tvar*, *head*, *type*, *name*);

bool
GCQ_DEQUEUED_NEXT_TYPED(*tvar*, *current*, *head*, *start*, *type*, *name*);

bool
GCQ_DEQUEUED_PREV_TYPED(*tvar*, *current*, *head*, *start*, *type*, *name*);

bool
GCQ_GOT_FIRST_COND(*var*, *head*, *cond*);

bool
GCQ_GOT_LAST_COND(*var*, *head*, *cond*);

bool
GCQ_GOT_NEXT_COND(*var*, *current*, *head*, *start*, *cond*);

bool
GCQ_GOT_PREV_COND(*var*, *current*, *head*, *start*, *cond*);

bool
GCQ_DEQUEUED_FIRST_COND(*var*, *head*, *cond*);

bool
GCQ_DEQUEUED_LAST_COND(*var*, *head*, *cond*);

bool
GCQ_DEQUEUED_NEXT_COND(*var*, *current*, *head*, *start*, *cond*);

bool
GCQ_DEQUEUED_PREV_COND(*var*, *current*, *head*, *start*, *cond*);

bool
GCQ_GOT_FIRST_COND_TYPED(*tvar*, *head*, *type*, *name*, *cond*);

bool
GCQ_GOT_LAST_COND_TYPED(*tvar*, *head*, *type*, *name*, *cond*);

bool
GCQ_GOT_NEXT_COND_TYPED(*tvar*, *current*, *head*, *start*, *type*, *name*, *cond*);

bool
GCQ_GOT_PREV_COND_TYPED(*tvar*, *current*, *head*, *start*, *type*, *name*, *cond*);

bool
GCQ_DEQUEUED_FIRST_COND_TYPED(*tvar*, *head*, *type*, *name*, *cond*);

bool
GCQ_DEQUEUED_LAST_COND_TYPED(*tvar*, *head*, *type*, *name*, *cond*);

bool
GCQ_DEQUEUED_NEXT_COND_TYPED(*tvar*, *current*, *head*, *start*, *type*, *name*, *cond*);

bool
GCQ_DEQUEUED_PREV_COND_TYPED(*tvar*, *current*, *head*, *start*, *type*, *name*, *cond*);

GCQ_FOREACH(*var*, *head*);

GCQ_FOREACH_REV(*var*, *head*);

GCQ_FOREACH_NVAR(*var*, *nvar*, *head*);

GCQ_FOREACH_NVAR_REV(*var*, *nvar*, *head*);

GCQ_FOREACH_RO(*var*, *nvar*, *head*);

GCQ_FOREACH_RO_REV(*var*, *nvar*, *head*);

GCQ_FOREACH_DEQUEUED(*var*, *nvar*, *head*);

GCQ_FOREACH_DEQUEUED_REV(*var*, *nvar*, *head*);

GCQ_FOREACH_TYPED(*var*, *head*, *tvar*, *type*, *name*);

GCQ_FOREACH_REV_TYPED(*var*, *head*, *tvar*, *type*, *name*);

GCQ_FOREACH_NVAR_TYPED(*var*, *nvar*, *head*, *tvar*, *type*, *name*);

GCQ_FOREACH_NVAR_REV_TYPED(*var*, *nvar*, *head*, *tvar*, *type*, *name*);

GCQ_FOREACH_RO_TYPED(*var*, *nvar*, *head*, *tvar*, *type*, *name*);

GCQ_FOREACH_RO_REV_TYPED(*var*, *nvar*, *head*, *tvar*, *type*, *name*);

GCQ_FOREACH_DEQUEUED_TYPED(*var*, *nvar*, *head*, *tvar*, *type*, *name*);

GCQ_FOREACH_DEQUEUED_REV_TYPED(*var*, *nvar*, *head*, *tvar*, *type*, *name*);

GCQ_FIND(*var*, *head*, *cond*);

GCQ_FIND_REV(*var*, *head*, *cond*);

GCQ_FIND_TYPED(*var*, *head*, *tvar*, *type*, *name*, *cond*);

GCQ_FIND_REV_TYPED(*var*, *head*, *tvar*, *type*, *name*, *cond*);

GCQ_ASSERT(*cond*);

DESCRIPTION

The generic circular queue is a doubly linked list designed for efficient merge operations and unconditional removal. All basic operations can be perfomed with or without use of a separate head, allowing easy replacement of any pointers where efficient removal is desired. The meaning of the data type will not change; direct use and defined operations can be mixed when convienient. The basic type is:

```
struct gcq {
        struct gcq *q_next;
        struct gcq *q_prev;
};
```

The structure must first be initialized such that the *q_next* and *q_prev* members point to the beginning of the *struct gcq*. This can be done with **gcq_init**() and **gcq_init_head**() or with constant initializers **GCQ_INIT**() and **GCQ_INIT_HEAD**(). A *struct gcq* should *never* be given NULL values.

The structure containing the *struct gcq* can be retrieved by pointer arithmetic in the **GCQ_ITEM**() macro. List traversal normally requires knowledge of the list head to safely retrieve list items.

Capitalized operation names are macros and should be assumed to cause multiple evaluation of arguments. TYPED variants of macros set a typed pointer variable instead of or in addition to *struct gcq* * arguments. Additional type specific inlines and macros around some GCQ operations can be useful.

A few assertions are provided when DIAGNOSTIC is defined in the kernel or _DIAGNOSTIC is defined in userland. If GCQ_USE_ASSERT is defined prior to header inclusions then **assert**() will be used for assertions and NDEBUG can be used to turn them off. **GCQ_ASSERT**() is a wrapper around the used assertion function. None of the operations accept NULL arguements, however this is not tested by assertion.

The head is separately named for type checking but contains only a *struct gcq*, a pointer to which can be retrieved via **gcq_hq**(). The reverse operation is performed by **gcq_head**(), turning the supplied *struct gcq* * into *struct gcq_head* *. **gcq_q**() returns its *struct gcq* * argument and is used for type checking in **GCQ_ITEM**(). There are no functions for retrieving the raw *q_prev* and *q_next* pointers as these are usually clearer when used directly (if at all).

gcq_remove() returns the element removed and is always a valid operation after initialization. **gcq_onlist**() returns false if the structure links to itself and true otherwise. **gcq_empty**() is the negation of this operation performed on a head. **gcq_linked**() tests if prev->q_next == next && next->q_prev == prev.

gcq_tie() ties *src* after *dst* such that that if the old lists are DST, DST2 and SRC, SRC2, the new list is DST, SRC, SRC2, DST2. If *dst* and *src* are on the same list then any elements between but not including *dst* and *src* are cut from the list. If dst == src then the result is the same as **gcq_remove**(). **gcq_tie**() is equivalent to **gcq_tie_after**() except that the latter must only be used with arguments on separate lists or not on lists and asserts that src != dst && dst->q_prev != src. **gcq_tie_before**() performs the same operation on dst->q_prev.

gcq_merge() moves any elements on list *src* (but not *src* itself) to list *dst*. It is normally used with two heads via **gcq_merge_head**() or **gcq_merge_tail**(). If GCQ_UNCONDITIONAL_MERGE is defined prior to header inclusion then the merge operations will always perform a tie then remove *src* from the new list, which may reduce code size slightly.

gcq_clear() initializes all elements currently linked with *q* and is normally used with a head as **gcq_remove_all**().

gcq_insert_after() and **gcq_insert_before**() are slightly optimized versions of **gcq_tie**() for the case where *off* is not on a list and include assertions to this effect, which are also useful to detect missing initialization. **gcq_insert_head**() and **gcq_insert_tail**() are the same operations applied to a head.

GCQ_GOT_FIRST() and **GCQ_GOT_LAST**() set *var* to a pointer to the first or last *struct gcq* in the list or NULL if the list is empty and return false if empty and true otherwise. The boolean return is to emphasise that it is not normally safe and useful to directly pass the raw first/next/etc. pointer to another function. The macros are written such that the NULL values will be optimized out if not otherwise used. DEQUEUED variants also remove the member from the list. COND variants take an additional condition that

is evaluated when the macro would otherwise return `true`. If the condition is false *var* or *tvar* is set to `NULL` and no dequeue is performed.

`GCQ_GOT_NEXT`() and variants take pointers to the current position, list head, and starting point as arguments. The list head will be skipped when it is reached unless it is equal to the starting point; upon reaching the starting point *var* will be set to `NULL` and the macro will return `false`. The next and prev macros also assert that *current* is on the list unless it is equal to *start*. These macros are the only provided method for iterating through the list from an arbitrary point. Traversal macros are only provided for list heads, however **`gcq_head`**() can be used to treat any item as a head.

Foreach variants contain an embedded `for` statement for iterating over a list. Those containing `REV` use the *q_prev* pointer for traversal, others use *q_next*. The plain **`GCQ_FOREACH`**() uses a single variable. `NVAR` variants save the next pointer at the top of the loop so that the current element can be removed without adjusting *var*. This is useful when *var* is passed to a function that might remove it but will not otherwise modify the list. When the head is reached both *var* and *nvar* elements are left pointing to the list head. `FOREACH` asserts that *var*, and `NVAR` asserts that *nvar* does not point to itself when starting the next loop. This assertion takes place after the variable is tested against the head so it is safe to remove all elements from the list. `RO` variants also set *nvar* but assert that the two variables are linked at the end of each iteration. This is useful when calling a function that is not supposed to remove the element passed. `DEQUEUED` varients are like `NVAR` but remove each element before the code block is executed. `TYPED` variants are equvalent to the untyped versions except that they take three extra arguments: a typed pointer, the type name, and the member name of the *`struct gcq`* used in this list. *tvar* is set to `NULL` when the head is reached.

`GCQ_FIND`() is a foreach loop that does nothing except break when the supplied condition is true. `REV` and `TYPED` variants are available.

SEE ALSO

`gcc`(1), `assert`(3), `_DIAGASSERT`(3), `queue`(3), `KASSERT`(9)

HISTORY

GCQ appeared in NetBSD 5.0.

NAME

getaddrinfo, **freeaddrinfo** — host and service name to socket address structure

SYNOPSIS

```
#include <netdb.h>

int
getaddrinfo(const char * restrict hostname,
    const char * restrict servname,
    const struct addrinfo * restrict hints,
    struct addrinfo ** restrict res);

void
freeaddrinfo(struct addrinfo *ai);
```

DESCRIPTION

The **getaddrinfo**() function is used to get a list of IP addresses and port numbers for host *hostname* and service *servname*. It is a replacement for and provides more flexibility than the gethostbyname(3) and getservbyname(3) functions.

The *hostname* and *servname* arguments are either pointers to NUL-terminated strings or the null pointer. An acceptable value for *hostname* is either a valid host name or a numeric host address string consisting of a dotted decimal IPv4 address or an IPv6 address. The *servname* is either a decimal port number or a service name listed in services(5). At least one of *hostname* and *servname* must be non-null.

hints is an optional pointer to a struct addrinfo, as defined by ⟨netdb.h⟩:

```
struct addrinfo {
        int ai_flags;         /* input flags */
        int ai_family;        /* protocol family for socket */
        int ai_socktype;      /* socket type */
        int ai_protocol;      /* protocol for socket */
        socklen_t ai_addrlen; /* length of socket-address */
        struct sockaddr *ai_addr; /* socket-address for socket */
        char *ai_canonname;   /* canonical name for service location */
        struct addrinfo *ai_next; /* pointer to next in list */
};
```

This structure can be used to provide hints concerning the type of socket that the caller supports or wishes to use. The caller can supply the following structure elements in *hints*:

ai_family The protocol family that should be used. When *ai_family* is set to PF_UNSPEC, it means the caller will accept any protocol family supported by the operating system.

ai_socktype Denotes the type of socket that is wanted: SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW. When *ai_socktype* is zero the caller will accept any socket type.

ai_protocol Indicates which transport protocol is desired, IPPROTO_UDP or IPPROTO_TCP. If *ai_protocol* is zero the caller will accept any protocol.

ai_flags *ai_flags* is formed by OR'ing the following values:

AI_CANONNAME If the AI_CANONNAME bit is set, a successful call to **getaddrinfo**() will return a NUL-terminated string containing the canonical name of the specified hostname in the *ai_canonname* element of the first addrinfo structure returned.

AI_NUMERICHOST If the AI_NUMERICHOST bit is set, it indicates that *hostname* should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.

AI_NUMERICSERV If the AI_NUMERICSERV bit is set, it indicates that the *servname* string contains a numeric port number. This is used to prevent service name resolution.

AI_PASSIVE If the AI_PASSIVE bit is set it indicates that the returned socket address structure is intended for use in a call to bind(2). In this case, if the *hostname* argument is the null pointer, then the IP address portion of the socket address structure will be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address.

If the AI_PASSIVE bit is not set, the returned socket address structure will be ready for use in a call to connect(2) for a connection-oriented protocol or connect(2), sendto(2), or sendmsg(2) if a connectionless protocol was chosen. The IP address portion of the socket address structure will be set to the loopback address if *hostname* is the null pointer and AI_PASSIVE is not set.

All other elements of the addrinfo structure passed via *hints* must be zero or the null pointer.

If *hints* is the null pointer, **getaddrinfo**() behaves as if the caller provided a struct addrinfo with *ai_family* set to PF_UNSPEC and all other elements set to zero or NULL.

After a successful call to **getaddrinfo**(), **res* is a pointer to a linked list of one or more addrinfo structures. The list can be traversed by following the *ai_next* pointer in each addrinfo structure until a null pointer is encountered. The three members *ai_family*, *ai_socktype*, and *ai_protocol* in each returned addrinfo structure are suitable for a call to socket(2). For each addrinfo structure in the list, the *ai_addr* member points to a filled-in socket address structure of length *ai_addrlen*.

This implementation of **getaddrinfo**() allows numeric IPv6 address notation with scope identifier, as documented in chapter 11 of draft-ietf-ipv6-scoping-arch-02.txt. By appending the percent character and scope identifier to addresses, one can fill the sin6_scope_id field for addresses. This would make management of scoped addresses easier and allows cut-and-paste input of scoped addresses.

At this moment the code supports only link-local addresses with the format. The scope identifier is hard-coded to the name of the hardware interface associated with the link (such as ne0). An example is "fe80::1%ne0", which means "fe80::1 on the link associated with the ne0 interface".

The current implementation assumes a one-to-one relationship between the interface and link, which is not necessarily true from the specification.

All of the information returned by **getaddrinfo**() is dynamically allocated: the addrinfo structures themselves as well as the socket address structures and the canonical host name strings included in the addrinfo structures.

Memory allocated for the dynamically allocated structures created by a successful call to **getaddrinfo**() is released by the **freeaddrinfo**() function. The *ai* pointer should be a addrinfo structure created by a call to **getaddrinfo**().

RETURN VALUES

getaddrinfo() returns zero on success or one of the error codes listed in gai_strerror(3) if an error occurs.

EXAMPLES

The following code tries to connect to “www.kame.net” service “http” via a stream socket. It loops through all the addresses available, regardless of address family. If the destination resolves to an IPv4 address, it will use an AF_INET socket. Similarly, if it resolves to IPv6, an AF_INET6 socket is used. Observe that there is no hardcoded reference to a particular address family. The code works even if **getaddrinfo**() returns addresses that are not IPv4/v6.

```
struct addrinfo hints, *res, *res0;
int error;
int s;
const char *cause = NULL;

memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo("www.kame.net", "http", &hints, &res0);
if (error) {
        errx(1, "%s", gai_strerror(error));
        /*NOTREACHED*/
}
s = -1;
for (res = res0; res; res = res->ai_next) {
        s = socket(res->ai_family, res->ai_socktype,
            res->ai_protocol);
        if (s < 0) {
                cause = "socket";
                continue;
        }

        if (connect(s, res->ai_addr, res->ai_addrlen) < 0) {
                cause = "connect";
                close(s);
                s = -1;
                continue;
        }

        break;  /* okay we got one */
}
if (s < 0) {
        err(1, "%s", cause);
        /*NOTREACHED*/
}
freeaddrinfo(res0);
```

The following example tries to open a wildcard listening socket onto service “http”, for all the address families available.

```
struct addrinfo hints, *res, *res0;
int error;
int s[MAXSOCK];
int nsock;
const char *cause = NULL;
```

```
memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
error = getaddrinfo(NULL, "http", &hints, &res0);
if (error) {
        errx(1, "%s", gai_strerror(error));
        /*NOTREACHED*/
}
nsock = 0;
for (res = res0; res && nsock < MAXSOCK; res = res->ai_next) {
        s[nsock] = socket(res->ai_family, res->ai_socktype,
            res->ai_protocol);
        if (s[nsock] < 0) {
                cause = "socket";
                continue;
        }

        if (bind(s[nsock], res->ai_addr, res->ai_addrlen) < 0) {
                cause = "bind";
                close(s[nsock]);
                continue;
        }
        (void) listen(s[nsock], 5);

        nsock++;
}
if (nsock == 0) {
        err(1, "%s", cause);
        /*NOTREACHED*/
}
freeaddrinfo(res0);
```

SEE ALSO

`bind`(2), `connect`(2), `send`(2), `socket`(2), `gai_strerror`(3), `gethostbyname`(3), `getnameinfo`(3), `getservbyname`(3), `resolver`(3), `hosts`(5), `resolv.conf`(5), `services`(5), `hostname`(7), `named`(8)

R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens, *Basic Socket Interface Extensions for IPv6*, RFC 3493, February 2003.

S. Deering, B. Haberman, T. Jinmei, E. Nordmark, and B. Zill, *IPv6 Scoped Address Architecture*, internet draft, draft-ietf-ipv6-scoping-arch-02.txt, work in progress material.

Craig Metz, "Protocol Independence Using the Sockets API", *Proceedings of the FREENIX track: 2000 USENIX annual technical conference*, June 2000.

STANDARDS

The **`getaddrinfo`**() function is defined by the IEEE Std 1003.1g-2000 (“POSIX.1”) draft specification and documented in `RFC 3493`, “Basic Socket Interface Extensions for IPv6”.

NAME

getarg, **arg_printusage** — collect command line options

SYNOPSIS

```
#include <getarg.h>

int
getarg(struct getargs *args, size_t num_args, int argc, char **argv,
    int *optind);

void
arg_printusage(struct getargs *args, size_t num_args, const char *progname,
    const char *extra_string);
```

DESCRIPTION

getarg() collects any command line options given to a program in an easily used way. **arg_printusage**() pretty-prints the available options, with a short help text.

args is the option specification to use, and it's an array of *struct getargs* elements. *num_args* is the size of *args* (in elements). *argc* and *argv* are the argument count and argument vector to extract option from. *optind* is a pointer to an integer where the index to the last processed argument is stored, it must be initialised to the first index (minus one) to process (normally 0) before the first call.

arg_printusage take the same *args* and *num_args* as getarg; *progname* is the name of the program (to be used in the help text), and *extra_string* is a string to print after the actual options to indicate more arguments. The usefulness of this function is realised only be people who has used programs that has help strings that doesn't match what the code does.

The *getargs* struct has the following elements.

```
struct getargs{
    const char *long_name;
    char short_name;
    enum { arg_integer,
           arg_string,
           arg_flag,
           arg_negative_flag,
           arg_strings,
           arg_double,
           arg_collect
    } type;
    void *value;
    const char *help;
    const char *arg_help;
};
```

long_name is the long name of the option, it can be NULL, if you don't want a long name. *short_name* is the characted to use as short option, it can be zero. If the option has a value the *value* field gets filled in with that value interpreted as specified by the *type* field. *help* is a longer help string for the option as a whole, if it's NULL the help text for the option is omitted (but it's still displayed in the synopsis). *arg_help* is a description of the argument, if NULL a default value will be used, depending on the type of the option:

arg_integer — the argument is a signed integer, and *value* should point to an *int*.

arg_string — the argument is a string, and *value* should point to a *char**.

arg_flag — the argument is a flag, and *value* should point to a *int*. It gets filled in with either zero or one, depending on how the option is given, the normal case being one. Note that if the option isn't given, the value isn't altered, so it should be initialised to some useful default.

arg_negative_flag — this is the same as *arg_flag* but it reverses the meaning of the flag (a given short option clears the flag), and the synopsis of a long option is negated.

arg_strings — the argument can be given multiple times, and the values are collected in an array; *value* should be a pointer to a *struct getarg_strings* structure, which holds a length and a string pointer.

arg_double — argument is a double precision floating point value, and *value* should point to a *double*.

arg_collect — allows more fine-grained control of the option parsing process. *value* should be a pointer to a *getarg_collect_info* structure:

```
typedef int (*getarg_collect_func)(int short_opt,
                                   int argc,
                                   char **argv,
                                   int *optind,
                                   int *optarg,
                                   void *data);

typedef struct getarg_collect_info {
    getarg_collect_func func;
    void *data;
} getarg_collect_info;
```

With the *func* member set to a function to call, and *data* to some application specific data. The parameters to the collect function are:

short_flag non-zero if this call is via a short option flag, zero otherwise

argc, *argv* the whole argument list

optind pointer to the index in argv where the flag is

optarg pointer to the index in argv[*optind] where the flag name starts

data application specific data

You can modify **optind*, and **optarg*, but to do this correct you (more or less) have to know about the inner workings of getarg.

You can skip parts of arguments by increasing **optarg* (you could implement the **-z***3* set of flags from **gzip** with this), or whole argument strings by increasing **optind* (let's say you want a flag **-c** *x y z* to specify a coordinate); if you also have to set **optarg* to a sane value.

The collect function should return one of ARG_ERR_NO_MATCH, ARG_ERR_BAD_ARG, ARG_ERR_NO_ARG, ENOMEM on error, zero otherwise.

For your convenience there is a function, **getarg_optarg**(), that returns the traditional argument string, and you pass it all arguments, sans data, that where

given to the collection function.

Don’t use this more this unless you absolutely have to.

Option parsing is similar to what `getopt` uses. Short options without arguments can be compressed (**`-xyz`** is the same as **`-x -y -z`**), and short options with arguments take these as either the rest of the argv-string or as the next option (**`-o`***`foo`*, or **`-o`** *`foo`*).

Long option names are prefixed with -- (double dash), and the value with a = (equal), **`--foo=`***`bar`*. Long option flags can either be specified as they are (**`--help`**), or with an (boolean parsable) option (**`--help=`***`yes`*, **`--help=`***`true`*, or similar), or they can also be negated (**`--no-help`** is the same as **`--help=`**no), and if you’re really confused you can do it multiple times (**`--no-no-help=`***`false`*, or even **`--no-no-help=`***`maybe`*).

EXAMPLE

```
#include <stdio.h>
#include <string.h>
#include <getarg.h>

char *source = "Ouagadougou";
char *destination;
int weight;
int include_catalog = 1;
int help_flag;

struct getargs args[] = {
    { "source",      's', arg_string,  &source,
      "source of shippment", "city" },
    { "destination", 'd', arg_string,  &destination,
      "destination of shippment", "city" },
    { "weight",      'w', arg_integer, &weight,
      "weight of shippment", "tons" },
    { "catalog",     'c', arg_negative_flag, &include_catalog,
      "include product catalog" },
    { "help",        'h', arg_flag, &help_flag }
};

int num_args = sizeof(args) / sizeof(args[0]); /* number of elements in args */

const char *progname = "ship++";

int
main(int argc, char **argv)
{
    int optind = 0;
    if (getarg(args, num_args, argc, argv, &optind)) {
        arg_printusage(args, num_args, progname, "stuff...");
        exit (1);
    }
    if (help_flag) {
        arg_printusage(args, num_args, progname, "stuff...");
        exit (0);
    }
```

```
    if (destination == NULL) {
        fprintf(stderr, "%s: must specify destination\n", progname);
        exit(1);
    }
    if (strcmp(source, destination) == 0) {
        fprintf(stderr, "%s: destination must be different from source\n");
        exit(1);
    }
    /* include more stuff here ... */
    exit(2);
}
```

The output help output from this program looks like this:

```
$ ship++ --help
Usage: ship++ [--source=city] [-s city] [--destination=city] [-d city]
   [--weight=tons] [-w tons] [--no-catalog] [-c] [--help] [-h] stuff...
-s city, --source=city      source of shippment
-d city, --destination=city destination of shippment
-w tons, --weight=tons      weight of shippment
-c, --no-catalog            include product catalog
```

BUGS

It should be more flexible, so it would be possible to use other more complicated option syntaxes, such as what `ps`(1), and `tar`(1), uses, or the AFS model where you can skip the flag names as long as the options come in the correct order.

Options with multiple arguments should be handled better.

Should be integreated with SL.

It’s very confusing that the struct you pass in is called getargS.

SEE ALSO

`getopt`(3)

NAME

getbootfile — get the name of the booted kernel file

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

const char *
getbootfile(void);
```

DESCRIPTION

getbootfile() returns a static pointer to the full path name of the file from which the current kernel was loaded. If it can not be determined, or the file is not ‘‘secure’’ (see `secure_path`(3)), `_PATH_UNIX` from ⟨`paths.h`⟩ is returned instead.

SEE ALSO

`secure_path`(3), `sysctl`(3)

HISTORY

The **getbootfile** function call appeared in FreeBSD 2.0 and NetBSD 1.6.

NAME

getbsize — get user block size

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

char *
getbsize(int *headerlenp, long *blocksizep);
```

DESCRIPTION

The **getbsize** function determines the user's preferred block size based on the value of the "BLOCKSIZE" environment variable; see environ(7) for details on its use and format.

The **getbsize** function returns a pointer to a NUL terminated string describing the block size, something like "1K-blocks". If the *headerlenp* parameter is not NULL the memory referenced by *headerlenp* is filled in with the length of the string (not including the terminating NUL). If the *blocksizep* parameter is not NULL the memory referenced by *blocksizep* is filled in with block size, in bytes.

If the user's block size is unreasonable, a warning message is written to standard error and the returned information reflects a block size of 512 bytes.

SEE ALSO

df(1), du(1), ls(1), systat(1), environ(7)

HISTORY

The **getbsize** function first appeared in 4.4BSD.

NAME

fgetc, **getc**, **getchar**, **getc_unlocked**, **getchar_unlocked**, **getw** — get next character or word from input stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

int
fgetc(FILE *stream);

int
getc(FILE *stream);

int
getchar();

int
getc_unlocked(FILE *stream);

int
getchar_unlocked();

int
getw(FILE *stream);
```

DESCRIPTION

The **fgetc**() function obtains the next input character (if present) from the stream pointed at by *stream*, or the next character pushed back on the stream via ungetc(3).

The **getc**() function acts essentially identically to **fgetc**(), but is a macro that expands in-line.

The **getchar**() function is equivalent to: getc with the argument stdin.

The **getc_unlocked**() and **getchar_unlocked**() functions provide functionality identical to that of **getc**() and **getchar**(), respectively, but do not perform implicit locking of the streams they operate on. In multi-threaded programs they may be used *only* within a scope in which the stream has been successfully locked by the calling thread using either flockfile(3) or ftrylockfile(3), and may later be released using funlockfile(3).

The **getw**() function obtains the next *int* (if present) from the stream pointed at by *stream*.

RETURN VALUES

If successful, these routines return the next requested object from the *stream*. If the stream is at end-of-file or a read error occurs, the routines return EOF. The routines feof(3) and ferror(3) must be used to distinguish between end-of-file and error. If an error occurs, the global variable *errno* is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return EOF until the condition is cleared with clearerr(3).

SEE ALSO

ferror(3), flockfile(3), fopen(3), fread(3), ftrylockfile(3), funlockfile(3), putc(3), ungetc(3)

STANDARDS

The **fgetc**(), **getc**() and **getchar**() functions conform to ANSI X3.159-1989 ("ANSI C89"). The **getc_unlocked**() and **getchar_unlocked**() functions conform to ISO/IEC 9945-1:1996 ("POSIX.1").

BUGS

Since `EOF` is a valid integer value, `feof`(3) and `ferror`(3) must be used to check for failure after calling **getw**(). The size and byte order of an *int* varies from one machine to another, and **getw**() is not recommended for portable applications.

NAME

getcwd, **getwd** — get working directory pathname

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

char *
getcwd(char *buf, size_t size);

char *
getwd(char *buf);
```

DESCRIPTION

The **getcwd**() function copies the absolute pathname of the current working directory into the memory referenced by *buf* and returns a pointer to *buf*. The *size* argument is the size, in bytes, of the array referenced by *buf*.

If *buf* is NULL, space is allocated as necessary to store the pathname. This space may later be free(3)'d.

The function **getwd**() is a compatibility routine which calls **getcwd**() with its *buf* argument and a size of MAXPATHLEN (as defined in the include file ⟨sys/param.h⟩). Obviously, *buf* should be at least MAXPATHLEN bytes in length.

These routines have traditionally been used by programs to save the name of a working directory for the purpose of returning to it. A much faster and less error-prone method of accomplishing this is to open the current directory ('.') and use the fchdir(2) function to return.

RETURN VALUES

Upon successful completion, a pointer to the pathname is returned. Otherwise a NULL pointer is returned and the global variable *errno* is set to indicate the error. In addition, **getwd**() copies the error message associated with *errno* into the memory referenced by *buf*.

ERRORS

The **getcwd**() function will fail if:

[EACCES]	Read or search permission was denied for a component of the pathname.
[EINVAL]	The *size* argument is zero.
[ENOENT]	A component of the pathname no longer exists.
[ENOMEM]	Insufficient memory is available.
[ERANGE]	The *size* argument is greater than zero but smaller than the length of the pathname plus 1.

SEE ALSO

chdir(2), fchdir(2), malloc(3), strerror(3)

STANDARDS

The **getcwd**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1"). The ability to specify a NULL pointer and have **getcwd**() allocate memory as necessary is an extension.

HISTORY

The **getwd**() function appeared in 4.0BSD.

SECURITY CONSIDERATIONS

As **getwd**() does not know the length of the supplied buffer, it is possible for a long (but valid) path to overflow the buffer and provide a means for an attacker to exploit the caller. **getcwd**() should be used in place of **getwd**() (the latter is only provided for compatibility purposes).

NAME

getdevmajor — get block or character device major number

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>
#include <sys/stat.h>

dev_t
getdevmajor(const char *name, mode_t type);
```

DESCRIPTION

The **getdevmajor**() function returns the major device number of the block or character device specified by *name* and a file type matching the one encoded in *type* which must be one of S_IFBLK or S_IFCHR.

RETURN VALUES

If no device matches the specified values, no information is available, or an error occurs, (dev_t)~0 is returned and *errno* is set to indicate the error.

EXAMPLES

To retrieve the major number for pty(4) slave devices (aka pts devices):

```
#include <stdlib.h>
#include <sys/stat.h>

dev_t pts;
pts = getdevmajor("pts", S_IFCHR);
```

To retrieve the major numbers for the block and character wd(4) devices:

```
#include <stdlib.h>
#include <sys/stat.h>

dev_t c, b;
c = getdevmajor("wd", S_IFCHR);
b = getdevmajor("wd", S_IFBLK);
```

ERRORS

The **getdevmajor**() function may fail and set *errno* for any of the errors specified for the library functions malloc(3) and sysctlbyname(3). In addition, the following errors may be reported:

[EINVAL] The value of the *major* argument is not S_IFCHR or S_IFBLK.

[ENOENT] The named device is not found.

SEE ALSO

stat(2), devname(3), malloc(3), sysctlbyname(3)

HISTORY

The **getdevmajor**() function call appeared in NetBSD 3.0.

NAME

getdirentries — get directory entries in a filesystem independent format

SYNOPSIS

```
#include <dirent.h>

int
getdirentries(int fd, char *buf, int nbytes, long *basep);
```

DESCRIPTION

This interface is provided for compatibility only and has been obsoleted by getdents(2).

getdirentries() reads directory entries from the directory referenced by the file descriptor *fd* into the buffer pointed to by *buf*, in a filesystem independent format. Up to *nbytes* of data will be transferred. *nbytes* must be greater than or equal to the block size associated with the file, see stat(2). Some filesystems may not support **getdirentries**() with buffers smaller than this size.

The data in the buffer is a series of *dirent* structures each containing the following entries:

```
unsigned long  d_fileno;
unsigned short d_reclen;
unsigned short d_namlen;
char           d_name[MAXNAMELEN + 1]; /* see below */
```

The *d_fileno* entry is a number which is unique for each distinct file in the filesystem. Files that are linked by hard links (see link(2)) have the same *d_fileno*. If *d_fileno* is zero, the entry refers to a deleted file.

The *d_reclen* entry is the length, in bytes, of the directory record.

The *d_namlen* entry specifies the length of the file name excluding the null byte. Thus the actual size of *d_name* may vary from 1 to MAXNAMELEN + 1.

The *d_name* entry contains a null terminated file name.

Entries may be separated by extra space. The *d_reclen* entry may be used as an offset from the start of a *dirent* structure to the next structure, if any.

The actual number of bytes transferred is returned. The current position pointer associated with *fd* is set to point to the next block of entries. The pointer may not advance by the number of bytes returned by **getdirentries**(). A value of zero is returned when the end of the directory has been reached.

getdirentries() writes the position of the block read into the location pointed to by *basep*. Alternatively, the current position pointer may be set and retrieved by lseek(2). The current position pointer should only be set to a value returned by lseek(2), a value returned in the location pointed to by *basep*, or zero.

RETURN VALUES

If successful, the number of bytes actually transferred is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

getdirentries() will fail if:

[EBADF] *fd* is not a valid file descriptor open for reading.

[EFAULT] Either *buf* or *basep* point outside the allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

SEE ALSO

lseek(2), open(2)

HISTORY

The **getdirentries**() function first appeared in 4.4BSD.

NAME

getdiskbyname, **setdisktab** — get generic disk description by its name

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/disklabel.h>
#include <disktab.h>

int
setdisktab(char *name);

struct disklabel *
getdiskbyname(const char *name);
```

DESCRIPTION

The **getdiskbyname**() function takes a disk name (e.g. `rm03`) and returns a prototype disk label describing its geometry information and the standard disk partition tables. All information is obtained from the `disktab`(5) file.

The **setdisktab**() function changes the default file name from `/etc/disktab` (aka `_PATH_DISKTAB`) to *name*.

RETURN VALUES

getdiskbyname() returns a null pointer if the entry is not found in the current `disktab` file.

setdisktab() returns 0 on success and −1 if *name* is a null pointer or points to an empty string.

FILES

`/etc/disktab` the default database of disk types.

SEE ALSO

`disklabel`(5), `disktab`(5), `disklabel`(8)

HISTORY

The **getdiskbyname**() function appeared in 4.3BSD.

The **setdisktab**() function appeared in NetBSD 1.4.

BUGS

The **getdiskbyname**() function leaves its results in an internal static object and returns a pointer to that object. Subsequent calls will modify the same object.

NAME

getdomainname, **setdomainname** — get/set domain name of current host

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
getdomainname(char *name, size_t namelen);

int
setdomainname(const char *name, size_t namelen);
```

DESCRIPTION

getdomainname() returns the standard domain name for the current processor, as previously set by **setdomainname**(). The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

setdomainname() sets the domain name of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

RETURN VALUES

If the call succeeds a value of 0 is returned. If the call fails, a value of −1 is returned and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* or *namelen* parameter gave an invalid address.
[EPERM]	The caller tried to set the domain name and was not the super-user.

SEE ALSO

gethostid(3), gethostname(3), sysctl(3)

HISTORY

The **getdomainname** function call appeared in 4.2BSD.

BUGS

Domain names are limited to MAXHOSTNAMELEN (from ⟨sys/param.h⟩) characters including null-termination, currently 256.

NAME

getdtablesize — get descriptor table size

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
getdtablesize(void);
```

DESCRIPTION

Each process has a fixed size descriptor table, which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The call **getdtablesize**() returns the size of this table.

SEE ALSO

close(2), dup(2), getrlimit(2), open(2), select(2), sysconf(3)

HISTORY

The **getdtablesize**() function call appeared in 4.2BSD.

NAME

getenv, **getenv_r**, **putenv**, **setenv**, **unsetenv** — environment variable functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

char *
getenv(const char *name);

int
getenv_r(const char *name, char *buf, size_t len);

int
setenv(const char *name, const char *value, int overwrite);

int
putenv(const char *string);

int
unsetenv(const char *name);
```

DESCRIPTION

These functions set, unset and fetch environment variables from the host *environment list*. For compatibility with differing environment conventions, the given arguments *name* and *value* may be appended and prepended, respectively, with an equal sign "=", except for **unsetenv**().

The **getenv**() function obtains the current value of the environment variable *name*. If the variable *name* is not in the current environment, a NULL pointer is returned.

The **getenv_r**() function obtains the current value of the environment variable *name* and copies it to *buf*. If *name* is not in the current environment, or the string length of the value of *name* is longer than *len* characters, then −1 is returned and *errno* is set to indicate the error.

The **setenv**() function inserts or resets the environment variable *name* in the current environment list. If the variable *name* does not exist in the list, it is inserted with the given *value*. If the variable does exist, the argument *overwrite* is tested; if *overwrite is* zero, the variable is not reset, otherwise it is reset to the given *value*.

The **putenv**() function takes an argument of the form ''name=value'' and is equivalent to:

```
setenv(name, value, 1);
```

The **unsetenv**() function deletes all instances of the variable name pointed to by *name* from the list.

RETURN VALUES

The functions **getenv_r**(), **setenv**(), **putenv**(), and **unsetenv**() return zero if successful; otherwise the global variable *errno* is set to indicate the error and a −1 is returned.

If **getenv**() is successful, the string returned should be considered read-only.

ERRORS

[EINVAL] The *name* argument to **unsetenv**() is a null pointer, points to an empty string, or points to a string containing an "=" character.

[ENOMEM] The function **setenv**() or **putenv**() failed because they were unable to allocate memory for the environment.

The function **getenv_r**() can return the following errors:

[ENOENT] The variable *name* was not found in the environment.

[ERANGE] The value of the named variable is too long to fit in the supplied buffer.

SEE ALSO

csh(1), sh(1), execve(2), environ(7)

STANDARDS

The **getenv**() function conforms to ANSI X3.159-1989 ("ANSI C89"). The **putenv**() function conforms to X/Open Portability Guide Issue 4 ("XPG4"). The **unsetenv**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The functions **setenv**() and **unsetenv**() appeared in Version 7 AT&T UNIX. The **putenv**() function appeared in 4.3BSD−Reno.

NAME

getfsent, **getfsspec**, **getfsfile**, **setfsent**, **endfsent** — get file system descriptor file entry

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <fstab.h>

struct fstab *
getfsent(void);

struct fstab *
getfsspec(const char *spec);

struct fstab *
getfsfile(const char *file);

int
setfsent(void);

void
endfsent(void);
```

DESCRIPTION

The **getfsent**(), **getfsspec**(), and **getfsfile**() functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the file system description file, ⟨`fstab.h`⟩.

```
struct fstab {
        char    *fs_spec;       /* block special device name */
        char    *fs_file;       /* file system path prefix */
        char    *fs_vfstype;    /* type of file system */
        char    *fs_mntops;     /* comma separated mount options */
        char    *fs_type;       /* rw, ro, sw, or xx */
        int     fs_freq;        /* dump frequency, in days */
        int     fs_passno;      /* pass number on parallel dump */
};
```

The fields have meanings described in `fstab`(5).

The **setfsent**() function opens the file (closing any previously opened file) or rewinds it if it is already open.

The **endfsent**() function closes the file.

The **getfsspec**() and **getfsfile**() functions search the entire file (opening it if necessary) for a matching special file name or file system file name.

For programs wishing to read the entire database, **getfsent**() reads the next entry (opening the file if necessary).

All entries in the file with a type field equivalent to `FSTAB_XX` are ignored.

RETURN VALUES

The **getfsent**(), **getfsspec**(), and **getfsfile**() functions return a null pointer (0) on `EOF` or error. The **setfsent**() function returns 0 on failure, 1 on success. The **endfsent**() function returns nothing.

FILES

`/etc/fstab`

SEE ALSO

`fstab`(5)

HISTORY

The **`getfsent`**() function appeared in 4.0BSD; the **`endfsent`**(), **`getfsfile`**(), **`getfsspec`**(), and **`setfsent`**() functions appeared in 4.3BSD.

BUGS

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it.

NAME

getgrent, **getgrent_r**, **getgrgid**, **getgrgid_r**, **getgrnam**, **getgrnam_r**, **setgroupent**, **setgrent**, **endgrent** — group database operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <grp.h>

struct group *
getgrent(void);

int
getgrent_r(struct group *grp, char *buffer, size_t buflen,
    struct group **result);

struct group *
getgrgid(gid_t gid);

int
getgrgid_r(gid_t gid, struct group *grp, char *buffer, size_t buflen,
    struct group **result);

struct group *
getgrnam(const char *name);

int
getgrnam_r(const char *name, struct group *grp, char *buffer, size_t buflen,
    struct group **result);

int
setgroupent(int stayopen);

void
setgrent(void);

void
endgrent(void);
```

DESCRIPTION

These functions operate on the group database which is described in group(5). Each line of the database is defined by the structure *group* found in the include file ⟨grp.h⟩:

```
struct group {
        char    *gr_name;       /* group name */
        char    *gr_passwd;     /* group password */
        gid_t   gr_gid;         /* group id */
        char    **gr_mem;       /* group members */
};
```

The functions **getgrnam**() and **getgrgid**() search the group database for the given group name pointed to by *name* or the group id pointed to by *gid*, respectively, returning the first one encountered. Identical group names or group ids may result in undefined behavior.

The **getgrent**() function sequentially reads the group database and is intended for programs that wish to step through the complete list of groups.

All three functions will open the group file for reading, if necessary.

The functions **getgrnam_r**(), **getgrgid_r**(), and **getgrent_r**() act like their non re-entrant counterparts respectively, updating the contents of *grp* and storing a pointer to that in *result*, and returning 0. Storage used by *grp* is allocated from *buffer*, which is *buflen* bytes in size. If the requested entry cannot be found, *result* will point to NULL and 0 will be returned. If an error occurs, a non-zero error number will be returned and *result* will point to NULL. Calling **getgrent_r**() from multiple threads will result in each thread reading a disjoint portion of the group database.

The **setgroupent**() function opens the file, or rewinds it if it is already open. If *stayopen* is non-zero, file descriptors are left open, significantly speeding functions subsequent calls. This functionality is unnecessary for **getgrent**() as it doesn't close its file descriptors by default. It should also be noted that it is dangerous for long-running programs to use this functionality as the group file may be updated.

The **setgrent**() function is equivalent to **setgroupent**() with an argument of zero.

The **endgrent**() function closes any open files.

RETURN VALUES

The functions **getgrgid**(), **getgrnam**(), and **getgrent**() return a valid pointer to a group structure on success and a NULL pointer if the entry was not found or an error occured. If an error occured, the global variable errno is set to indicate the nature of the failure.

The functions **getgrgid_r**(), **getgrnam_r**(), and **getgrent_r**() return 0 on success or entry not found, and non-zero on failure, setting the global variable errno to indicate the nature of the failure.

The **setgroupent**() function returns the value 1 if successful, otherwise the value 0 is returned, setting the global variable errno to indicate the nature of the failure.

The **endgrent**() and **setgrent**() functions have no return value.

ERRORS

The following error codes may be set in *errno* for **getgrent**, **getgrent_r**, **getgrnam**, **getgrnam_r**, **getgrgid**, **getgrgid_r**, and **setgroupent**:

[EIO] An I/O error has occurred.

[EINTR] A signal was caught during the database search.

[EMFILE] The limit on open files for this process has been reached.

[ENFILE] The system limit on open files has been reached.

The following error code may be set in *errno* for **getgrent_r**, **getgrnam_r**, and **getgrgid_r**:

[ERANGE] The resulting *struct group* does not fit in the space defined by buffer and buflen

Other errno values may be set depending on the specific database backends.

FILES

/etc/group group database file

SEE ALSO

getpwent(3), group(5), nsswitch.conf(5)

STANDARDS

The **getgrgid**() and **getgrnam**() functions conform to ISO/IEC 9945-1:1990 (“POSIX.1”). The **getgrgid_r**() and **getgrnam_r**() functions conform to IEEE Std 1003.1c-1995 (“POSIX.1”). The **endgrent**(), **getgrent**(), and **setgrent**() functions conform to X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”) and IEEE Std 1003.1-2004 " (“POSIX.1”) (XSI extension).

HISTORY

The functions **endgrent**(), **getgrent**(), **getgrgid**(), **getgrnam**(), and **setgrent**() appeared in Version 7 AT&T UNIX. The functions **setgrfile**() and **setgroupent**() appeared in 4.3BSD−Reno. The functions **getgrgid_r**() and **getgrnam_r**() appeared in NetBSD 3.0.

COMPATIBILITY

The historic function **setgrfile**(), which allowed the specification of alternative group databases, has been deprecated and is no longer available.

BUGS

The functions **getgrent**(), **getgrgid**(), **getgrnam**(), **setgroupent**() and **setgrent**() leave their results in an internal static object and return a pointer to that object. Subsequent calls to the same function will modify the same object.

The functions **getgrent**(), **endgrent**(), **setgroupent**(), and **setgrent**() are fairly useless in a networked environment and should be avoided, if possible. **getgrent**() makes no attempt to suppress duplicate information if multiple sources are specified in `nsswitch.conf`(5)

NAME

getgrouplist, **getgroupmembership**, — calculate group access list

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
getgrouplist(const char *name, gid_t basegid, gid_t *groups, int *ngroups);

int
getgroupmembership(const char *name, gid_t basegid, gid_t *groups,
    int maxgrp, int *ngroups);
```

DESCRIPTION

The **getgrouplist**() and **getgroupmembership**() functions read through the group database and calculate the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password database.

The resulting group list is returned in the integer array pointed to by *groups*.

For **getgrouplist**(), the caller specifies the size of the *groups* array in the integer pointed to by *ngroups*.

For **getgroupmembership**(), the caller specifies the size of the *groups* array in *maxgrp*.

The actual number of groups found is returned in *ngroups*.

Duplicate group ids will be suppressed from the result.

RETURN VALUES

The **getgrouplist**() and **getgroupmembership**() functions return 0 if successful, and return −1 if the size of the group list is too small to hold all the user's groups. In the latter case, the *groups* array will be filled with as many groups as will fit and *ngroups* will contain the total number of groups found.

FILES

`/etc/group` group membership list

SEE ALSO

`setgroups`(2), `initgroups`(3), `group`(5)

HISTORY

The **getgrouplist**() function first appeared in 4.4BSD. The **getgroupmembership**() function first appeared in NetBSD 3.0 to address an API deficiency in **getgrouplist**().

BUGS

The **getgrouplist**() function uses the routines based on `getgrent`(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to **getgrouplist**().

NAME

gethostbyname, **gethostbyname2**, **gethostbyaddr**, **gethostent**, **sethostent**, **endhostent**, **herror**, **hstrerror** — get network host entry

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <netdb.h>
extern int h_errno;

struct hostent *
gethostbyname(const char *name);

struct hostent *
gethostbyname2(const char *name, int af);

struct hostent *
gethostbyaddr(const char *addr, socklen_t len, int type);

struct hostent *
gethostent(void);

void
sethostent(int stayopen);

void
endhostent(void);

void
herror(const char *string);

const char *
hstrerror(int err);
```

DESCRIPTION

The **gethostbyname**(), **gethostbyname2**() and **gethostbyaddr**() functions each return a pointer to an object with the following structure describing an internet host.

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses from name server */
};
#define h_addr  h_addr_list[0] /* address, for backward compatibility */
```

The members of this structure are:

h_name Official name of the host.

h_aliases A NULL-terminated array of alternative names for the host.

h_addrtype The type of address being returned; currently always AF_INET.

h_length The length, in bytes, of the address.

h_addr_list A NULL-terminated array of network addresses for the host. Host addresses are returned in network byte order.

h_addr The first address in *h_addr_list*; this is for backward compatibility.

In the case of **gethostbyname**() and **gethostbyname2**(), the host is specified by name, or using a string representation of a numeric address. In the case of **gethostbyaddr**(), the host is specified using a binary representation of an address.

The returned *struct hostent* structure may contain the result of a simple string to binary conversion, information obtained from the domain name resolver (see resolver(3)), broken-out fields from a line in /etc/hosts, or database entries supplied by the yp(8) system. The order of the lookups is controlled by the 'hosts' entry in nsswitch.conf(5).

When using the domain name resolver, **gethostbyname**() and **gethostbyname2**() will search for the named host in the current domain and its parents unless the name ends in a dot. If the name contains no dot, and if the environment variable "HOSTALIASES" contains the name of an alias file, the alias file will first be searched for an alias matching the input name. See hostname(7) for the domain search procedure and the alias file format.

The **gethostbyname2**() function is an evolution of **gethostbyname**() which is intended to allow lookups in address families other than AF_INET, for example AF_INET6. Currently the *af* argument must be specified as AF_INET or AF_INET6, else the function will return NULL after having set *h_errno* to NETDB_INTERNAL.

The **gethostent**() function reads the next line of the /etc/hosts file, opening the file if necessary.

The **sethostent**() function may be used to request the use of a connected TCP socket for queries. If the *stayopen* flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to **gethostbyname**(), **gethostbyname2**(), or **gethostbyaddr**(). Otherwise, queries are performed using UDP datagrams.

The **endhostent**() function closes the TCP connection.

The **herror**() function writes a message to the diagnostic output consisting of the string parameter *s*, the constant string ": ", and a message corresponding to the value of *h_errno*.

The **hstrerror**() function returns a string which is the message text corresponding to the value of the *err* parameter.

FILES

/etc/hosts

DIAGNOSTICS

Error return status from **gethostbyent**(), **gethostbyname**(), **gethostbyname2**(), and **gethostbyaddr**() is indicated by return of a null pointer. The external integer *h_errno* may then be checked to see whether this is a temporary failure or an invalid or unknown host. The routine **herror**() can be used to print an error message describing the failure. If its argument *string* is non-NULL, it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

The variable *h_errno* can have the following values:

HOST_NOT_FOUND No such host is known.

TRY_AGAIN This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.

NO_RECOVERY Some unexpected server failure was encountered. This is a non-recoverable error.

NO_DATA The requested name is valid but does not have an IP address; this is not a temporary error. This means that the name is known to the name server but there is no address associated with this name. Another type of request to the name server using this domain name will result in an answer; for example, a mail-forwarder may be registered for this domain.

SEE ALSO

resolver(3), hosts(5), nsswitch.conf(5), hostname(7), named(8)

HISTORY

The **herror**() function appeared in 4.3BSD. The **endhostent**(), **gethostbyaddr**(), **gethostbyname**(), **gethostent**(), and **sethostent**() functions appeared in 4.2BSD. The **gethostbyname2**() function first appeared in bind-4.9.4. IPv6 support was implemented in WIDE Hydrangea IPv6 protocol stack kit.

CAVEATS

If the search routines specified in nsswitch.conf(5) decide to read the /etc/hosts file, **gethostbyname**(), **gethostbyname2**(), and **gethostbyaddr**() will read the next line of the file, re-opening the file if necessary.

The **sethostent**() function opens and/or rewinds the file /etc/hosts. If the *stayopen* argument is non-zero, the file will not be closed after each call to **gethostbyname**(), **gethostbyname2**(), **gethostbyaddr**(), or **gethostent**().

The **endhostent**() function closes the file.

BUGS

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet address format is currently understood.

The **gethostent**() does not currently follow the search order specified in nsswitch.conf(5) and only reads the /etc/hosts file.

NAME

gethostid, **sethostid** — get/set unique identifier of current host

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

long
gethostid(void);

int
sethostid(long hostid);
```

DESCRIPTION

sethostid() establishes a 32-bit identifier for the current processor that is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine. This call is allowed only to the super-user and is normally performed at boot time.

gethostid() returns the 32-bit identifier for the current processor.

This function has been deprecated. The hostid should be set or retrieved by use of `sysctl`(3).

SEE ALSO

`gethostname`(3), `sysctl`(3), `sysctl`(8)

HISTORY

The **gethostid**() and **sethostid**() syscalls appeared in 4.2BSD and were dropped in 4.4BSD.

BUGS

32 bits for the identifier is too small.

NAME

gethostname, **sethostname** — get/set name of current host

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
gethostname(char *name, size_t namelen);

int
sethostname(const char *name, size_t namelen);
```

DESCRIPTION

gethostname() returns the standard host name for the current processor, as previously set by **sethostname**(). The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

sethostname() sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

RETURN VALUES

If the call succeeds a value of 0 is returned. If the call fails, a value of −1 is returned and an error code is placed in the global location *errno*.

ERRORS

If the **gethostname**() or **sethostname**() functions fail, they will set *errno* for any of the errors specified for the routine `sysctl`(3).

SEE ALSO

`gethostid`(3), `sysctl`(3), `sysctl`(8)

STANDARDS

The **gethostname**() function conforms to X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”).

HISTORY

The **gethostname** function call appeared in 4.2BSD.

BUGS

Host names are limited to `MAXHOSTNAMELEN` (from ⟨`sys/param.h`⟩) characters including null-termination, currently 256.

NAME

getifaddrs — get interface addresses

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <ifaddrs.h>

int
getifaddrs(struct ifaddrs **ifap);

void
freeifaddrs(struct ifaddrs *ifp);
```

DESCRIPTION

The **getifaddrs**() function stores a reference to a linked list of the network interfaces on the local machine in the memory referenced by *ifap*. The list consists of **ifaddrs** structures, as defined in the include file ⟨ifaddrs.h⟩. The **ifaddrs** structure contains at least the following entries:

```
struct ifaddrs   *ifa_next;         /* Pointer to next struct */
char             *ifa_name;         /* Interface name */
u_int             ifa_flags;        /* Interface flags */
struct sockaddr  *ifa_addr;         /* Interface address */
struct sockaddr  *ifa_netmask;      /* Interface netmask */
struct sockaddr  *ifa_broadaddr;    /* Interface broadcast address */
struct sockaddr  *ifa_dstaddr;      /* P2P interface destination */
void             *ifa_data;              /* Address specific data */
```

The ifa_next field contains a pointer to the next structure on the list. This field is NULL in last structure on the list.

The ifa_name field contains the interface name.

The ifa_flags field contains the interface flags, as set by ifconfig(8) utility.

The ifa_addr field references either the address of the interface or the link level address of the interface, if one exists, otherwise it is NULL. (The sa_family field of the ifa_addr field should be consulted to determine the format of the ifa_addr address.)

The ifa_netmask field references the netmask associated with ifa_addr, if one is set, otherwise it is NULL.

The ifa_broadaddr field, which should only be referenced for non-P2P interfaces, references the broadcast address associated with ifa_addr, if one exists, otherwise it is NULL.

The ifa_dstaddr field references the destination address on a P2P interface, if one exists, otherwise it is NULL.

The ifa_data field references address family specific data. For AF_LINK addresses it contains a pointer to the *struct if_data* (as defined in include file ⟨net/if.h⟩) which contains various interface attributes and statistics. For all other address families, it contains a pointer to the *struct ifa_data* (as defined in include file ⟨net/if.h⟩) which contains per-address interface statistics.

The data returned by **getifaddrs**() is dynamically allocated and should be freed using **freeifaddrs**() when no longer needed.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

The **getifaddrs**() may fail and set *errno* for any of the errors specified for the library routines ioctl(2), socket(2), malloc(3) or sysctl(3).

SEE ALSO

ioctl(2), socket(2), sysctl(3), networking(4), ifconfig(8)

HISTORY

The **getifaddrs** implementation first appeared in BSD/OS.

BUGS

If both ⟨net/if.h⟩ and ⟨ifaddrs.h⟩ are being included, ⟨net/if.h⟩ *must* be included before ⟨ifaddrs.h⟩.

NAME

getlabelsector, **getlabeloffset** — get the sector number and offset of the disklabel

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

daddr_t
getlabelsector(void);

off_t
getlabeloffset(void);
```

DESCRIPTION

The **getlabelsector**() and **getlabeloffset**() functions return values which describe the exact on-disk location of the disklabel(5) on the current system, or −1 on error. These functions supercede the hardcoded LABELSECTOR and LABELOFFSET definitions previously used to derive the location of the disklabel(5).

SEE ALSO

sysctl(3), disklabel(5)

HISTORY

The **getlabelsector**() and **getlabeloffset**() functions appeared in NetBSD 2.0.

NAME

getlastlogx, **getutmp**, **getutmpx**, **updlastlogx**, **updwtmpx**, **utmpxname** — user accounting database functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <utmpx.h>

struct lastlogx *
getlastlogx(const char *fname, uid_t uid, struct lastlogx *ll);

void
getutmp(const struct utmpx *ux, struct utmp *u);

void
getutmpx(const struct utmp *u, struct utmpx *ux);

int
updlastlogx(const char *fname, uid_t uid, struct lastlogx *ll);

int
updwtmpx(const char *file, const struct utmpx *utx);

int
utmpxname(const char *fname);
```

DESCRIPTION

The **getlastlogx**() function looks up the entry for the user with user id *uid* in the lastlogx(5) file given by *fname* and returns it in *ll*. If the provided *ll* is NULL, the necessary space will be allocated by **getlastlogx**() and should be **free**()d by the caller.

The **getutmp**() function fills out the entries in the struct utmp *u* with the data provided in the struct utmpx *ux*. **getutmpx**() does the opposite, filling out the entries in the struct utmpx *ux* with the data provided in the struct utmp *u*, and initializing all the unknown fields to 0. The sole exception is the *ut_type* field, which will be initialized to USER_PROCESS.

The **updlastlogx**() function tries to update the information for the user with the user id *uid* in the lastlogx(5) file given by *fname* with the data supplied in *ll*. A *struct lastlogx* is defined like this:

```
struct lastlogx {
        struct timeval ll_tv;           /* time entry was created */
        char ll_line[_UTX_LINESIZE];    /* tty name */
        char ll_host[_UTX_HOSTSIZE];    /* host name */
        struct sockaddr_storage ll_ss;  /* address where entry was made from */
};
```

All the fields should be filled out by the caller.

The **updwtmpx**() function updates the wtmpx(5) file *file* with the utmpx(5) entry *utx*.

The **utmpxname**() function sets the default utmpx(5) database file name to *fname*.

RETURN VALUES

getlastlogx() returns the found entry on success, or NULL if it could not open the database, could not find an entry matching *uid* in there, or could not allocate the necessary space (in case *ll* was NULL).

utmpxname() returns 1 on success, or 0 if the supplied file name was too long or did not end with ‘x’.

updlastlogx() and **updwtmpx**() return 0 on success, or −1 in case the database or file respectively could not be opened or the data not written into it.

SEE ALSO

endutxent(3), loginx(3), utmpx(5)

HISTORY

The functions **getutmp**(), **getutmpx**(), **updwtmpx**(), and **utmpxname**() first appeared in Solaris. **getlastlogx** and **updlastlogx** first appeared in NetBSD 2.0.

NAME

getloadavg — get system load averages

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
getloadavg(double loadavg[], int nelem);
```

DESCRIPTION

The **getloadavg**() function returns the number of processes in the system run queue averaged over various periods of time. Up to *nelem* samples are retrieved and assigned to successive elements of *loadavg*[]. The system imposes a maximum of 3 samples, representing averages over the last 1, 5, and 15 minutes, respectively.

DIAGNOSTICS

If the load average was unobtainable, −1 is returned; otherwise, the number of samples actually retrieved is returned.

SEE ALSO

uptime(1), kvm_getloadavg(3), sysctl(3)

HISTORY

The **getloadavg**() function appeared in 4.3BSD−Reno.

NAME

getmaxpartitions — get the maximum number of partitions allowed per disk

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
getmaxpartitions(void);
```

DESCRIPTION

getmaxpartitions() returns the number of partitions that are allowed per disk on the system, or −1 in case of an error, setting the global *errno* variable. The possible values for *errno* are the same as in sysctl(3).

SEE ALSO

getrawpartition(3), sysctl(3)

HISTORY

The **getmaxpartitions** function call appeared in NetBSD 1.2.

NAME

getmntinfo — get information about mounted file systems

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/statvfs.h>

int
getmntinfo(struct statvfs **mntbufp, int flags);
```

DESCRIPTION

The **getmntinfo**() function returns an array of *statvfs* structures describing each currently mounted file system (see statvfs(2)).

The **getmntinfo**() function passes its *flags* parameter transparently to getvfsstat(2).

RETURN VALUES

On successful completion, **getmntinfo**() returns a count of the number of elements in the array. The pointer to the array is stored into *mntbufp*.

If an error occurs, zero is returned and the external variable *errno* is set to indicate the error. Although the pointer *mntbufp* will be unmodified, any information previously returned by **getmntinfo**() will be lost.

ERRORS

The **getmntinfo**() function may fail and set *errno* for any of the errors specified for the library routines getvfsstat(2) or malloc(3).

SEE ALSO

getvfsstat(2), mount(2), statvfs(2), mount(8)

HISTORY

The **getmntinfo**() function first appeared in 4.4BSD. It was converted from using getfsstat(2) to getvfsstat(2) in NetBSD 3.0.

BUGS

The **getmntinfo**() function writes the array of structures to an internal static object and returns a pointer to that object. Subsequent calls to **getmntinfo**() will modify the same object.

The memory allocated by **getmntinfo**() cannot be free(3)'d by the application.

NAME

getmntopts — scan mount options

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <mntopts.h>

mntoptparse_t
getmntopts(const char *options, const struct mntopt *mopts, int *flagp,
    int *altflagp);

const char *
getmntoptstr(mntoptparse_t mp, const char *opt);

long
getmntoptnum(mntoptparse_t mp, const char *opt);

void
freemntopts(mntoptparse_t mp);
```

DESCRIPTION

The **getmntopts**() function takes a comma separated option list and a list of valid option names, and computes the bitmask corresponding to the requested set of options.

The string *options* is broken down into a sequence of comma separated tokens. Each token is looked up in the table described by *mopts* and the bits in the word referenced by either *flagp* or *altflagp* (depending on the m_altloc field of the option's table entry) are updated. The flag words are not initialized by **getmntopts**(). The table, *mopts*, has the following format:

```
struct mntopt {
        const char *m_option;  /* option name */
        int m_inverse;         /* negative option, e.g., "dev" */
        int m_flag;            /* bit to set, e.g., MNT_RDONLY */
        int m_altloc;          /* use altflagp rather than flagp */
};
```

The members of this structure are:

m_option the option name, for example "suid".

m_inverse tells **getmntopts**() that the name has the inverse meaning of the bit. For example, "suid" is the string, whereas the mount flag is MNT_NOSUID. In this case, the sense of the string and the flag are inverted, so the *m_inverse* flag should be set.

m_flag the value of the bit to be set or cleared in the flag word when the option is recognized. The bit is set when the option is discovered, but cleared if the option name was preceded by the letters "no". The *m_inverse* flag causes these two operations to be reversed.

m_altloc the bit should be set or cleared in *altflagp* rather than *flagp*.

Each of the user visible MNT_ flags has a corresponding MOPT_ macro which defines an appropriate struct mntopt entry. To simplify the program interface and ensure consistency across all programs, a general purpose macro, MOPT_STDOPTS, is defined which contains an entry for all the generic VFS options. In addition, the macros MOPT_FORCE and MOPT_UPDATE exist to enable the MNT_FORCE and MNT_UPDATE flags to be set. Finally, the table must be terminated by an entry with a NULL first element.

The **getmntoptstr**() function returns the string value of the named option, if such a value was set it the option string.

The **getmntoptnum**() returns the long value of the named option, if such a value was set it the option string. It prints an error message and exits if the value was not set, or could not be converted from a string to a long.

The **freemntopts**() frees the storage used by **getmntopts**().

EXAMPLES

Most commands will use the standard option set. Local filesystems which support the MNT_UPDATE flag, would also have an MOPT_UPDATE entry. This can be declared and used as follows:

```
#include <mntopts.h>

static const struct mntopt mopts[] = {
        MOPT_STDOPTS,
        MOPT_UPDATE,
        { NULL }
};

long val;
mntflags = mntaltflags = 0;
mntoptparse_t mp;
if ((mp = getmntopts(options, mopts, &mntflags, &mntaltflags)) == NULL)
        err(1, NULL);
val = getmntoptnum(mp, "rsize");
freemntopts(mp);
```

RETURN VALUE

getmntopts() returns NULL if an error occurred. Note that some bits may already have been set in *flagp* and *altflagp* even if NULL is returned. **getmntoptstr**() returns NULL if the option does not have an argument, or the option string. **getmntoptnum**() returns −1 if an error occurred.

DIAGNOSTICS

If the external integer variable *getmnt_silent* is zero then the **getmntopts**() function displays an error message and exits if an unrecognized option is encountered. By default *getmnt_silent* is zero.

SEE ALSO

err(3), mount(8)

HISTORY

The **getmntopts**() function appeared in 4.4BSD. It was moved to the utilities library and enhanced to retrieve option values in NetBSD 2.0.

NAME

getnameinfo — socket address structure to hostname and service name

SYNOPSIS

```
#include <netdb.h>

int
getnameinfo(const struct sockaddr * restrict sa, socklen_t salen,
    char * restrict host, size_t hostlen, char * restrict serv,
    size_t servlen, int flags);
```

DESCRIPTION

The **getnameinfo**() function is used to convert a `sockaddr` structure to a pair of host name and service strings. It is a replacement for and provides more flexibility than the `gethostbyaddr`(3) and `getservbyport`(3) functions and is the converse of the `getaddrinfo`(3) function.

The `sockaddr` structure *sa* should point to either a `sockaddr_in` or `sockaddr_in6` structure (for IPv4 or IPv6 respectively) that is *salen* bytes long.

The host and service names associated with *sa* are stored in *host* and *serv* which have length parameters *hostlen* and *servlen*. The maximum value for *hostlen* is `NI_MAXHOST` and the maximum value for *servlen* is `NI_MAXSERV`, as defined by ⟨`netdb.h`⟩. If a length parameter is zero, no string will be stored. Otherwise, enough space must be provided to store the host name or service string plus a byte for the NUL terminator.

The *flags* argument is formed by **OR**'ing the following values:

`NI_NOFQDN`	A fully qualified domain name is not required for local hosts. The local part of the fully qualified domain name is returned instead.
`NI_NUMERICHOST`	Return the address in numeric form, as if calling `inet_ntop`(3), instead of a host name.
`NI_NAMEREQD`	A name is required. If the host name cannot be found in DNS and this flag is set, a non-zero error code is returned. If the host name is not found and the flag is not set, the address is returned in numeric form.
NI_NUMERICSERV	The service name is returned as a digit string representing the port number.
NI_DGRAM	Specifies that the service being looked up is a datagram service, and causes `getservbyport`(3) to be called with a second argument of "udp" instead of its default of "tcp". This is required for the few ports (512−514) that have different services for UDP and TCP.

This implementation allows numeric IPv6 address notation with scope identifier, as documented in chapter 11 of draft-ietf-ipv6-scoping-arch-02.txt. IPv6 link-local address will appear as a string like "`fe80::1%ne0`". Refer to `getaddrinfo`(3) for more information.

RETURN VALUES

getnameinfo() returns zero on success or one of the error codes listed in `gai_strerror`(3) if an error occurs.

EXAMPLES

The following code tries to get a numeric host name, and service name, for a given socket address. Observe that there is no hardcoded reference to a particular address family.

```
struct sockaddr *sa;   /* input */
char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), sbuf,
    sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV)) {
        errx(1, "could not get numeric hostname");
        /*NOTREACHED*/
}
printf("host=%s, serv=%s\n", hbuf, sbuf);
```

The following version checks if the socket address has a reverse address mapping:

```
struct sockaddr *sa;   /* input */
char hbuf[NI_MAXHOST];

if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), NULL, 0,
    NI_NAMEREQD)) {
        errx(1, "could not resolve hostname");
        /*NOTREACHED*/
}
printf("host=%s\n", hbuf);
```

SEE ALSO

`gai_strerror`(3), `getaddrinfo`(3), `gethostbyaddr`(3), `getservbyport`(3), `inet_ntop`(3), `resolver`(3), `hosts`(5), `resolv.conf`(5), `services`(5), `hostname`(7), `named`(8)

R. Gilligan, S. Thomson, J. Bound, and W. Stevens, *Basic Socket Interface Extensions for IPv6*, RFC 2553, March 1999.

S. Deering, B. Haberman, T. Jinmei, E. Nordmark, and B. Zill, *IPv6 Scoped Address Architecture*, internet draft, draft-ietf-ipv6-scoping-arch-02.txt, work in progress material.

Craig Metz, "Protocol Independence Using the Sockets API", *Proceedings of the FREENIX track: 2000 USENIX annual technical conference*, June 2000.

STANDARDS

The **`getnameinfo`**() function is defined by the IEEE Std 1003.1g-2000 ("POSIX.1") draft specification and documented in **RFC 2553**, "Basic Socket Interface Extensions for IPv6".

CAVEATS

`getnameinfo`() can return both numeric and FQDN forms of the address specified in *`sa`*. There is no return value that indicates whether the string returned in *`host`* is a result of binary to numeric-text translation (like `inet_ntop`(3)), or is the result of a DNS reverse lookup. Because of this, malicious parties could set up a PTR record as follows:

```
1.0.0.127.in-addr.arpa. IN PTR  10.1.1.1
```

and trick the caller of **`getnameinfo`**() into believing that *`sa`* is `10.1.1.1` when it is actually `127.0.0.1`.

To prevent such attacks, the use of `NI_NAMEREQD` is recommended when the result of **`getnameinfo`**() is used for access control purposes:

```
struct sockaddr *sa;
socklen_t salen;
char addr[NI_MAXHOST];
```

```
struct addrinfo hints, *res;
int error;

error = getnameinfo(sa, salen, addr, sizeof(addr),
    NULL, 0, NI_NAMEREQD);
if (error == 0) {
        memset(&hints, 0, sizeof(hints));
        hints.ai_socktype = SOCK_DGRAM;       /*dummy*/
        hints.ai_flags = AI_NUMERICHOST;
        if (getaddrinfo(addr, "0", &hints, &res) == 0) {
                /* malicious PTR record */
                freeaddrinfo(res);
                printf("bogus PTR record\n");
                return -1;
        }
        /* addr is FQDN as a result of PTR lookup */
} else {
        /* addr is numeric string */
        error = getnameinfo(sa, salen, addr, sizeof(addr),
            NULL, 0, NI_NUMERICHOST);
}
```

BUGS

The implementation of **getnameinfo**() is not thread-safe.

NAME

getnetconfig, **setnetconfig**, **endnetconfig**, **getnetconfigent**, **freenetconfigent**, **nc_perror**, **nc_sperror** — get network configuration database entry

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <netconfig.h>

struct netconfig *
getnetconfig(void *handlep);

void *
setnetconfig(void);

int
endnetconfig(void *handlep);

struct netconfig *
getnetconfigent(const char *netid);

void
freenetconfigent(struct netconfig *netconfigp);

void
nc_perror(const char *msg);

char *
nc_sperror(void);
```

DESCRIPTION

The library routines described on this page provide the application access to the system network configuration database, `/etc/netconfig`.

```
struct netconfig {
        char *nc_netid;              /* Network ID */
        unsigned long nc_semantics;  /* Semantics */
        unsigned long nc_flag;       /* Flags */
        char *nc_protofmly;          /* Protocol family */
        char *nc_proto;              /* Protocol name */
        char *nc_device;             /* Network device pathname */
        unsigned long nc_nlookups;   /* Number of directory lookup libs */
        char **nc_lookups;           /* Names of the libraries */
};
```

getnetconfig() returns a pointer to the current entry in the `netconfig` database, formatted as a struct netconfig. Successive calls will return successive netconfig entries in the netconfig database. **getnetconfig**() can be used to search the entire netconfig file. **getnetconfig**() returns `NULL` at the end of the file. *handlep* is the handle obtained through **setnetconfig**().

A call to **setnetconfig**() has the effect of ‘‘binding’’ to or ‘‘rewinding’’ the netconfig database. **setnetconfig**() must be called before the first call to **getnetconfig**() and may be called at any other time. **setnetconfig**() need not be called before a call to **getnetconfigent**(). **setnetconfig**() returns a unique handle to be used by **getnetconfig**().

endnetconfig() should be called when processing is complete to release resources for reuse. *handlep* is the handle obtained through **setnetconfig**(). Programmers should be aware, however, that the last call to **endnetconfig**() frees all memory allocated by **getnetconfig**() for the struct netconfig data structure. **endnetconfig**() may not be called before **setnetconfig**().

getnetconfigent() returns a pointer to the netconfig structure corresponding to *netid*. It returns NULL if *netid* is invalid (that is, does not name an entry in the netconfig database).

freenetconfigent() frees the netconfig structure pointed to by *netconfigp* (previously returned by **getnetconfigent**()).

nc_perror() prints a message to the standard error indicating why any of the above routines failed. The message is prepended with the string *msg* and a colon. A newline character is appended at the end of the message.

nc_sperror() is similar to **nc_perror**() but instead of sending the message to the standard error, will return a pointer to a string that contains the error message.

nc_perror() and **nc_sperror**() can also be used with the *NETPATH* access routines defined in getnetpath(3).

RETURN VALUES

setnetconfig() returns a unique handle to be used by **getnetconfig**(). In the case of an error, **setnetconfig**() returns NULL and **nc_perror**() or **nc_sperror**() can be used to print the reason for failure.

getnetconfig() returns a pointer to the current entry in the netconfig database, formatted as a struct netconfig. **getnetconfig**() returns NULL at the end of the file, or upon failure.

endnetconfig() returns 0 on success and -1 on failure (for example, if **setnetconfig**() was not called previously).

On success, **getnetconfigent**() returns a pointer to the struct netconfig structure corresponding to *netid*; otherwise it returns NULL.

nc_sperror() returns a pointer to a buffer which contains the error message string. This buffer is overwritten on each call. In multithreaded applications, this buffer is implemented as thread-specific data.

FILES

/etc/netconfig

SEE ALSO

getnetpath(3), netconfig(5)

NAME

getnetent, **getnetbyaddr**, **getnetbyname**, **setnetent**, **endnetent** — get network entry

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <netdb.h>

struct netent *
getnetent();

struct netent *
getnetbyname(const char *name);

struct netent *
getnetbyaddr(uint32_t net, int type);

setnetent(int stayopen);

endnetent();
```

DESCRIPTION

The **getnetent**(), **getnetbyname**(), and **getnetbyaddr**() functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base as described in networks(5).

```
struct  netent {
        char            *n_name;        /* official name of net */
        char            **n_aliases;    /* alias list */
        int             n_addrtype;     /* net number type */
        uint32_t        n_net;          /* net number */
};
```

The members of this structure are:

n_name The official name of the network.

n_aliases A zero terminated list of alternative names for the network.

n_addrtype The type of the network number returned; currently only AF_INET.

n_net The network number. Network numbers are returned in machine byte order.

The **getnetent**() function reads the next line of the file, opening the file if necessary.

The **setnetent**() function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getnetbyname**() or **getnetbyaddr**().

The **endnetent**() function closes the file.

The **getnetbyname**() function and **getnetbyaddr**() sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

DIAGNOSTICS

Null pointer (0) returned on `EOF` or error.

SEE ALSO

`networks`(5), `nsswitch.conf`(5)

HISTORY

The **`getnetent`**(), **`getnetbyaddr`**(), **`getnetbyname`**(), **`setnetent`**(), and **`endnetent`**() functions appeared in 4.2BSD.

BUGS

The data space used by these functions is static; if future use requires the data, it should be copied before any subsequent calls to these functions overwrite it. Only Internet network numbers are currently understood. Expecting network numbers to fit in no more than 32 bits is probably naive.

NAME

getnetgrent, **innetgr**, **setnetgrent**, **endnetgrent** — netgroup database operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <netgroup.h>

int
getnetgrent(const char **host, const char **user, const char **domain);

int
innetgr(const char *netgroup, const char *host, const char *user,
    const char *domain);

void
setnetgrent(const char *netgroup);

void
endnetgrent(void);
```

DESCRIPTION

These functions operate on the netgroup database file which is described in netgroup(5).

The database defines a set of netgroups, each made up of one or more triples:

```
(host, user, domain)
```

that defines a combination of host, user and domain. Any of the three fields may be specified as ‘‘wildcards’’ that match any string.

The function **getnetgrent**() sets the three pointer arguments to the strings of the next member of the current netgroup. If any of the string pointers are NULL that field is considered a wildcard.

The functions **setnetgrent**() and **endnetgrent**() set the current netgroup and terminate the current netgroup respectively. If **setnetgrent**() is called with a different netgroup than the previous call, an implicit **endnetgrent**() is implied. **setnetgrent**() also sets the offset to the first member of the netgroup.

The function **innetgr**() searches for a match of all fields within the specified group. If any of the **host**, **user**, or **domain** arguments are NULL those fields will match any string value in the netgroup member.

RETURN VALUES

The function **getnetgrent**() returns 0 for ‘‘no more netgroup members’’ and 1 otherwise. The function **innetgr**() returns 1 for a successful match and 0 otherwise. The functions **setnetgrent**() and **endnetgrent**() have no return value.

FILES

/etc/netgroup netgroup database file

SEE ALSO

netgroup(5), nsswitch.conf(5)

BUGS

The function **getnetgrent**() returns pointers to dynamically allocated data areas that are free'd when the function **endnetgrent**() is called.

NAME

getnetpath, **setnetpath**, **endnetpath** — get /etc/netconfig entry corresponding to NETPATH component

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <netconfig.h>

struct netconfig *
getnetpath(void *handlep);

void *
setnetpath(void);

int
endnetpath(void *handlep);
```

DESCRIPTION

The routines described in this page provide the application access to the system network configuration database, `/etc/netconfig`, as it is "filtered" by the NETPATH environment variable (see `environ`(7)). See `getnetconfig`(3) for other routines that also access the network configuration database directly. The NETPATH variable is a list of colon-separated network identifiers.

getnetpath() returns a pointer to the netconfig database entry corresponding to the first valid NETPATH component. The netconfig entry is formatted as a struct netconfig. On each subsequent call, **getnetpath**() returns a pointer to the netconfig entry that corresponds to the next valid NETPATH component. **getnetpath**() can thus be used to search the netconfig database for all networks included in the NETPATH variable. When NETPATH has been exhausted, **getnetpath**() returns NULL.

A call to **setnetpath**() "binds" to or "rewinds" NETPATH. **setnetpath**() must be called before the first call to **getnetpath**() and may be called at any other time. It returns a handle that is used by **getnetpath**().

getnetpath() silently ignores invalid NETPATH components. A NETPATH component is invalid if there is no corresponding entry in the netconfig database.

If the NETPATH variable is unset, **getnetpath**() behaves as if NETPATH were set to the sequence of "default" or "visible" networks in the netconfig database, in the order in which they are listed.

endnetpath() may be called to "unbind" from NETPATH when processing is complete, releasing resources for reuse. Programmers should be aware, however, that **endnetpath**() frees all memory allocated by **getnetpath**() for the struct netconfig data structure.

RETURN VALUES

setnetpath() returns a handle that is used by **getnetpath**(). In case of an error, **setnetpath**() returns NULL.

endnetpath() returns 0 on success and -1 on failure (for example, if **setnetpath**() was not called previously). **nc_perror**() or **nc_sperror**() can be used to print out the reason for failure. See `getnetconfig`(3).

When first called, **getnetpath**() returns a pointer to the netconfig database entry corresponding to the first valid NETPATH component. When NETPATH has been exhausted, **getnetpath**() returns NULL.

SEE ALSO

`getnetconfig`(3), `netconfig`(5), `environ`(7)

NAME

getopt — get option character from command line argument list

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

extern char *optarg;
extern int optind;
extern int optopt;
extern int opterr;
extern int optreset;

int
getopt(int argc, char * const argv[], const char *optstring);
```

DESCRIPTION

The **getopt**() function incrementally parses a command line argument list *argv* and returns the next *known* option character. An option character is *known* if it has been specified in the string of accepted option characters, *optstring*.

The option string *optstring* may contain the following elements: individual characters, and characters followed by a colon to indicate an option argument is to follow. For example, an option string "x" recognizes an option "**-x**", and an option string "x:" recognizes an option and argument "**-x** *argument*". It does not matter to **getopt**() if a following argument has leading whitespace.

On return from **getopt**(), *optarg* points to an option argument, if it is anticipated, and the variable *optind* contains the index to the next *argv* argument for a subsequent call to **getopt**(). The variable *optopt* saves the last *known* option character returned by **getopt**().

The variables *opterr* and *optind* are both initialized to 1. The *optind* variable may be set to another value before a set of calls to **getopt**() in order to skip over more or less argv entries.

In order to use **getopt**() to evaluate multiple sets of arguments, or to evaluate a single set of arguments multiple times, the variable *optreset* must be set to 1 before the second and each additional set of calls to **getopt**(), and the variable *optind* must be reinitialized.

The **getopt**() function returns −1 when the argument list is exhausted. The interpretation of options in the argument list may be cancelled by the option "--" (double dash) which causes **getopt**() to signal the end of argument processing and return −1. When all options have been processed (i.e., up to the first non-option argument), **getopt**() returns −1.

RETURN VALUES

The **getopt**() function returns the next known option character in *optstring*. If **getopt**() encounters a character not found in *optstring* or if it detects a missing option argument, it returns '?' (question mark). If *optstring* has a leading ':' then a missing option argument causes ':' to be returned instead of '?'. In either case, the variable *optopt* is set to the character that caused the error. The **getopt**() function returns −1 when the argument list is exhausted.

EXAMPLES

```
extern char *optarg;
extern int optind;
int bflag, ch, fd;
```

```
bflag = 0;
while ((ch = getopt(argc, argv, "bf:")) != -1) {
        switch (ch) {
        case 'b':
                bflag = 1;
                break;
        case 'f':
                if ((fd = open(optarg, O_RDONLY, 0)) < 0) {
                        (void)fprintf(stderr,
                            "myname: %s: %s\n", optarg, strerror(errno));
                        exit(1);
                }
                break;
        case '?':
        default:
                usage();
        }
}
argc -= optind;
argv += optind;
```

DIAGNOSTICS

If the **getopt**() function encounters a character not found in the string *optstring* or detects a missing option argument it writes an error message to *stderr* and returns '?'. Setting *opterr* to a zero will disable these error messages. If *optstring* has a leading ':' then a missing option argument causes a ':' to be returned in addition to suppressing any error messages.

Option arguments are allowed to begin with '-'; this is reasonable but reduces the amount of error checking possible.

SEE ALSO

getopt(1), getopt_long(3), getsubopt(3)

STANDARDS

The *optreset* variable was added to make it possible to call the **getopt**() function multiple times. This is an extension to the IEEE Std 1003.2 ("POSIX.2") specification.

HISTORY

The **getopt**() function appeared in 4.3BSD.

BUGS

The **getopt**() function was once specified to return EOF instead of −1. This was changed by IEEE Std 1003.2-1992 ("POSIX.2") to decouple **getopt**() from ⟨stdio.h⟩.

A single dash ('-') may be specified as a character in *optstring*, however it should *never* have an argument associated with it. This allows **getopt**() to be used with programs that expect '-' as an option flag. This practice is wrong, and should not be used in any current development. It is provided for backward compatibility *only*. Care should be taken not to use '-' as the first character in *optstring* to avoid a semantic conflict with GNU **getopt**(), which assigns different meaning to an *optstring* that begins with a '-'. By default, a single dash causes **getopt**() to return −1.

It is also possible to handle digits as option letters. This allows **getopt**() to be used with programs that expect a number ("−3") as an option. This practice is wrong, and should not be used in any current devel-

opment. It is provided for backward compatibility *only*. The following code fragment works in most cases.

```
int ch;
long length;
char *p;

while ((ch = getopt(argc, argv, "0123456789")) != -1) {
        switch (ch) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
                p = argv[optind - 1];
                if (p[0] == '-' && p[1] == ch && !p[2])
                        length = ch - '0';
                else
                        length = strtol(argv[optind] + 1, NULL, 10);
                break;
        }
}
```

NAME

getopt_long — get long options from command line argument list

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <getopt.h>

int
getopt_long(int argc, char * const *argv, const char *optstring,
    struct option *long_options, int *index);
```

DESCRIPTION

The **getopt_long**() function is similar to getopt(3) but it accepts options in two forms: words and characters. The **getopt_long**() function provides a superset of the functionality of getopt(3). **getopt_long**() can be used in two ways. In the first way, every long option understood by the program has a corresponding short option, and the option structure is only used to translate from long options to short options. When used in this fashion, **getopt_long**() behaves identically to getopt(3). This is a good way to add long option processing to an existing program with the minimum of rewriting.

In the second mechanism, a long option sets a flag in the *option* structure passed, or will store a pointer to the command line argument in the *option* structure passed to it for options that take arguments. Additionally, the long option's argument may be specified as a single argument with an equal sign, e.g.

```
myprogram --myoption=somevalue
```

When a long option is processed the call to **getopt_long**() will return 0. For this reason, long option processing without shortcuts is not backwards compatible with getopt(3).

It is possible to combine these methods, providing for long options processing with short option equivalents for some options. Less frequently used options would be processed as long options only.

Abbreviated long option names are accepted when **getopt_long**() processes long options if the abbreviation is unique. An exact match is always preferred for a defined long option.

The **getopt_long**() call requires a structure to be initialized describing the long options. The structure is:

```
struct option {
        char *name;
        int has_arg;
        int *flag;
        int val;
};
```

The *name* field should contain the option name without the leading double dash.

The *has_arg* field should be one of:

no_argument	no argument to the option is expect.
required_argument	an argument to the option is required.
optional_argument	an argument to the option may be presented.

If *flag* is not NULL, then the integer pointed to by it will be set to the value in the *val* field. If the *flag* field is NULL, then the *val* field will be returned. Setting *flag* to NULL and setting *val* to the corresponding short option will make this function act just like getopt(3).

If the *index* field is not NULL, the integer it points to will be set to the index of the long option in the *long_options* array.

The last element of the *long_options* array has to be filled with zeroes (see **EXAMPLES** section).

EXAMPLES

```
     extern char *optarg;
     extern int optind;
     int bflag, ch, fd;
     int daggerset;

     /* options descriptor */
     static struct option longopts[] = {
            { "buffy",     no_argument,          0,            'b' },
            { "fluoride",  required_argument,    0,            'f' },
            { "daggerset", no_argument,          &daggerset,   1 },
            { NULL,        0,                    NULL,         0 }
     };

     bflag = 0;
     while ((ch = getopt_long(argc, argv, "bf:", longopts, NULL)) != -1)
            switch (ch) {
            case 'b':
                    bflag = 1;
                    break;
            case 'f':
                    if ((fd = open(optarg, O_RDONLY, 0)) < 0) {
                            (void)fprintf(stderr,
                                "myname: %s: %s\n", optarg, strerror(errno));
                            exit(1);
                    }
                    break;
            case 0:
                    if(daggerset) {
                            fprintf(stderr,"Buffy will use her dagger to "
                                           "apply fluoride to dracula's teeth\n");
                    }
                    break;
            case '?':
            default:
                    usage();
     }
     argc -= optind;
     argv += optind;
```

IMPLEMENTATION DIFFERENCES

This section describes differences to the GNU implementation found in glibc-2.1.3:

- handling of - as first char of option string in presence of environment variable POSIXLY_CORRECT:

  GNU ignores POSIXLY_CORRECT and returns non-options as arguments to option '\1'.

NetBSD honors POSIXLY_CORRECT and stops at the first non-option.

- handling of :: in options string in presence of POSIXLY_CORRECT:

  Both GNU and NetBSD ignore POSIXLY_CORRECT here and take :: to mean the preceding option takes an optional argument.

- return value in case of missing argument if first character (after + or -) in option string is not ':':

  GNU returns '?'

  NetBSD returns ':' (since NetBSD's getopt does).

- handling of --a in getopt:

  GNU parses this as option '-', option 'a'.

  NetBSD parses this as '--', and returns −1 (ignoring the a). (Because the original getopt does.)

- setting of optopt for long options with flag != NULL:

  GNU sets optopt to val.

  NetBSD sets optopt to 0 (since val would never be returned).

- handling of -W with W; in option string in getopt (not getopt_long):

  GNU causes a segfault.

  NetBSD returns −1, with optind pointing past the argument of -W (as if '-W arg' were '--arg', and thus '--' had been found).

- setting of optarg for long options without an argument that are invoked via -W (W; in option string):

  GNU sets optarg to the option name (the argument of -W).

  NetBSD sets optarg to NULL (the argument of the long option).

- handling of -W with an argument that is not (a prefix to) a known long option (W; in option string):

  GNU returns -W with optarg set to the unknown option.

  NetBSD treats this as an error (unknown option) and returns '?' with optopt set to 0 and optarg set to NULL (as GNU's man page documents).

- The error messages are different.

- NetBSD does not permute the argument vector at the same points in the calling sequence as GNU does. The aspects normally used by the caller (ordering after −1 is returned, value of optind relative to current positions) are the same, though. (We do fewer variable swaps.)

SEE ALSO

getopt(3)

HISTORY

The **getopt_long**() function first appeared in GNU libiberty. The first NetBSD implementation appeared in 1.5.

BUGS

The implementation can completely replace getopt(3), but right now we are using separate code.

The *argv* argument is not really const.

NAME

getpagesize — get system page size

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
getpagesize(void);
```

DESCRIPTION

This interface is obsoleted by `sysconf`**(3).**

getpagesize() returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The page size is a *system* page size and may not be the same as the underlying hardware page size.

SEE ALSO

`pagesize`(1), `sbrk`(2), `sysconf`(3)

HISTORY

The **getpagesize** function call appeared in 4.2BSD.

NAME

getpass — get a password

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <pwd.h>
#include <unistd.h>

char *
getpass(const char *prompt);
```

DESCRIPTION

The **getpass**() function displays a prompt to, and reads in a password from, `/dev/tty`. If this file is not accessible, **getpass** displays the prompt on the standard error output and reads from the standard input.

The password may be up to _PASSWORD_LEN (currently 128) characters in length. Any additional characters and the terminating newline character are discarded.

getpass turns off character echoing while reading the password.

RETURN VALUES

getpass returns a pointer to the null terminated password.

FILES

`/dev/tty`

SEE ALSO

`crypt`(3)

HISTORY

A **getpass** function appeared in Version 7 AT&T UNIX.

BUGS

The **getpass** function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to **getpass** will modify the same object.

SECURITY CONSIDERATIONS

The calling process should zero the password as soon as possible to avoid leaving the cleartext password visible in the process's address space.

NAME

getpeereid — get the effective credentials of a UNIX-domain peer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int
getpeereid(int s, uid_t *euid, gid_t *egid);
```

DESCRIPTION

The **getpeereid**() function returns the effective user and group IDs of the peer connected to a UNIX-domain socket. The argument *s* must be a UNIX-domain socket (unix(4)) of type SOCK_STREAM on which either connect(2) has been called, or one returned from accept(2) after bind(2) and listen(2) have been called. If non-NULL, the effective used ID is placed in *euid*, and the effective group ID in *egid*.

The credentials returned to the accept(2) caller are those of its peer at the time it called connect(2); the credentials returned to the connect(2) caller are those of its peer at the time it called bind(2). This mechanism is reliable; there is no way for either side to influence the credentials returned to its peer except by calling the appropriate system call (i.e., either connect(2) or bind(2)) under different effective credentials.

One common use of this routine is for a UNIX-domain server to verify the credentials of its client. Likewise, the client can verify the credentials of the server.

IMPLEMENTATION NOTES

On NetBSD, **getpeereid**() is implemented in terms of the LOCAL_PEEREID unix(4) socket option.

RETURN VALUES

The **getpeereid**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getpeereid**() function fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The argument *s* does not refer to a socket on which connect(2) have been called nor one returned from listen(2).
[EINVAL]	The argument *s* does not refer to a socket of type SOCK_STREAM, or the kernel returned invalid data.

SEE ALSO

connect(2), getpeername(2), getsockname(2), getsockopt(2), listen(2), unix(4)

HISTORY

The **getpeereid**() function appeared in NetBSD 5.0.

NAME

getprogname, **setprogname** — get/set the name of the current program

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

const char *
getprogname(void);

void
setprogname(const char *name);
```

DESCRIPTION

These utility functions get and set the current program's name as used by various error-reporting functions.

getprogname() returns the name of the current program. This function is typically useful when generating error messages or other diagnostic output. If the program name has not been set, **getprogname**() will return NULL.

setprogname() sets the name of the current program to be the last pathname component of the *name* argument. It should be invoked at the start of the program, using the *argv[0]* passed into the program's **main**() function. A pointer into the string pointed to by the *name* argument is kept as the program name. Therefore, the string pointed to by *name* should not be modified during the rest of the program's operation.

A program's name can only be set once, and in NetBSD that is actually done by program start-up code that is run before **main**() is called. Therefore, in NetBSD, calling **setprogname**() from **main**() has no effect. However, it does serve to increase the portability of the program: on other operating systems, **getprogname**() and **setprogname**() may be implemented by a portability library, and a call to **setprogname**() allows that library to know the program name without modifications to that system's program start-up code.

SEE ALSO

err(3), setproctitle(3)

HISTORY

The **getprogname** and **setprogname** function calls appeared in NetBSD 1.6.

RESTRICTIONS

The string returned by **getprogname**() is supplied by the invoking process and should not be trusted by setuid or setgid programs.

NAME

getprotoent, **getprotobynumber**, **getprotobyname**, **setprotoent**, **endprotoent** — get protocol entry

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <netdb.h>

struct protoent *
getprotoent();

struct protoent *
getprotobyname(const char *name);

struct protoent *
getprotobynumber(int proto);

setprotoent(int stayopen);

endprotoent();
```

DESCRIPTION

The **getprotoent**(), **getprotobyname**(), and **getprotobynumber**() functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, `/etc/protocols`.

```
struct  protoent {
        char    *p_name;        /* official name of protocol */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol number */
};
```

The members of this structure are:

p_name The official name of the protocol.

p_aliases A zero terminated list of alternative names for the protocol.

p_proto The protocol number.

The **getprotoent**() function reads the next line of the file, opening the file if necessary.

The **setprotoent**() function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getprotobyname**() or **getprotobynumber**().

The **endprotoent**() function closes the file.

The **getprotobyname**() function and **getprotobynumber**() sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until `EOF` is encountered.

RETURN VALUES

Null pointer (0) returned on `EOF` or error.

FILES

`/etc/protocols`

SEE ALSO

`protocols`(5)

HISTORY

The **getprotoent**(), **getprotobynumber**(), **getprotobyname**(), **setprotoent**(), and **endprotoent**() functions appeared in 4.2BSD.

BUGS

These functions use a static data space; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet protocols are currently understood.

NAME

getpwent, **getpwent_r**, **getpwnam**, **getpwnam_r**, **getpwuid**, **getpwuid_r**, **setpassent**, **setpwent**, **endpwent** — password database operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <pwd.h>

struct passwd *
getpwent(void);

int
getpwent_r(struct passwd *pw, char *buffer, size_t buflen,
    struct passwd **result);

struct passwd *
getpwnam(const char *name);

int
getpwnam_r(const char *name, struct passwd *pw, char *buffer, size_t buflen,
    struct passwd **result);

struct passwd *
getpwuid(uid_t uid);

int
getpwuid_r(uid_t uid, struct passwd *pw, char *buffer, size_t buflen,
    struct passwd **result);

int
setpassent(int stayopen);

void
setpwent(void);

void
endpwent(void);
```

DESCRIPTION

These functions operate on the password database which is described in passwd(5). Each entry in the database is defined by the structure *passwd* found in the include file ⟨pwd.h⟩:

```
struct passwd {
        char    *pw_name;       /* user name */
        char    *pw_passwd;     /* encrypted password */
        uid_t   pw_uid;         /* user uid */
        gid_t   pw_gid;         /* user gid */
        time_t  pw_change;      /* password change time */
        char    *pw_class;      /* user login class */
        char    *pw_gecos;      /* general information */
        char    *pw_dir;        /* home directory */
        char    *pw_shell;      /* default shell */
        time_t  pw_expire;      /* account expiration */
};
```

The functions **getpwnam**() and **getpwuid**() search the password database for the given user name pointed to by *name* or user id pointed to by *uid* respectively, always returning the first one encountered. Identical user names or user ids may result in undefined behavior.

The **getpwent**() function sequentially reads the password database and is intended for programs that wish to process the complete list of users.

The functions **getpwnam_r**(), **getpwuid_r**(), and **getpwent_r**() act like their non re-entrant counterparts, updating the contents of *pw* and storing a pointer to that in *result*, and returning `0`. Storage used by *pw* is allocated from *buffer*, which is *buflen* bytes in size. If the requested entry cannot be found, *result* will point to `NULL` and `0` will be returned. If an error occurs, a non-zero error number will be returned and *result* will point to `NULL`. Calling **getpwent_r**() from multiple threads will result in each thread reading a disjoint portion of the password database.

The **setpassent**() function accomplishes two purposes. First, it causes **getpwent**() to "rewind" to the beginning of the database. Additionally, if *stayopen* is non-zero, file descriptors are left open, significantly speeding up subsequent accesses for all of the functions. (This latter functionality is unnecessary for **getpwent**() as it doesn't close its file descriptors by default.)

It is dangerous for long-running programs to keep the file descriptors open as the database will become out of date if it is updated while the program is running.

The **setpwent**() function is equivalent to **setpassent**() with an argument of zero.

The **endpwent**() function closes any open files.

These functions have been written to "shadow" the password file, e.g. allow only certain programs to have access to the encrypted password. If the process which calls them has an effective uid of 0, the encrypted password will be returned, otherwise, the password field of the returned structure will point to the string '*'.

RETURN VALUES

The functions **getpwent**(), **getpwnam**(), and **getpwuid**(), return a valid pointer to a passwd structure on success and a `NULL` pointer if the entry was not found or an error occured. If an error occured, the global variable `errno` is set to indicate the nature of the failure. The **setpassent**() function returns 0 on failure, setting the global variable `errno` to indicate the nature of the failure, and 1 on success. The **endpwent**() and **setpwent**() functions have no return value. The functions **getpwnam_r**(), **getpwuid_r**(), and **getpwent_r**() return `0` on success or entry not found, and non-zero on failure, setting the global variable `errno` to indicate the nature of the failure.

ERRORS

The following error codes may be set in *errno* for **getpwent**, **getpwent_r**, **getpwnam**, **getpwnam_r**, **getpwuid**, **getpwuid_r**, and **setpassent**:

`[EIO]` An I/O error has occurred.

`[EINTR]` A signal was caught during the database search.

`[EMFILE]` The limit on open files for this process has been reached.

`[ENFILE]` The system limit on open files has been reached.

The following error code may be set in *errno* for **getpwent_r**, **getpwnam_r**, and **getpwuid_r**:

`[ERANGE]` The resulting *struct passwd* does not fit in the space defined by `buffer` and `buflen`

Other `errno` values may be set depending on the specific database backends.

FILES

`/etc/pwd.db`	The insecure password database file
`/etc/spwd.db`	The secure password database file
`/etc/master.passwd`	The current password file
`/etc/passwd`	A Version 7 format password file

SEE ALSO

`getlogin`(2), `getgrent`(3), `nsswitch.conf`(5), `passwd`(5), `passwd.conf`(5), `pwd_mkdb`(8), `vipw`(8)

STANDARDS

The **getpwnam**() and **getpwuid**(), functions conform to ISO/IEC 9945-1:1990 ("POSIX.1"). The **getpwnam_r**() and **getpwuid_r**() functions conform to IEEE Std 1003.1c-1995 ("POSIX.1"). The **endpwent**(), **getpwent**(), and **setpwent**() functions conform to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2") and IEEE Std 1003.1-2004 " ("POSIX.1") (XSI extension).

HISTORY

The **getpwent**, **getpwnam**, **getpwuid**, **setpwent**, and **endpwent** functions appeared in Version 7 AT&T UNIX. The **setpassent** function appeared in 4.3BSD−Reno. The functions **getpwnam_r**() and **getpwuid_r**() appeared in NetBSD 3.0.

BUGS

The functions **getpwent**(), **getpwnam**(), and **getpwuid**(), leave their results in an internal static object and return a pointer to that object. Subsequent calls to any of these functions will modify the same object.

The functions **getpwent**(), **endpwent**(), **setpassent**(), and **setpwent**() are fairly useless in a networked environment and should be avoided, if possible. **getpwent**() makes no attempt to suppress duplicate information if multiple sources are specified in `nsswitch.conf`(5).

COMPATIBILITY

The historic function **setpwfile**() which allowed the specification of alternative password databases, has been deprecated and is no longer available.

NAME

getrawpartition — get the system “raw” partition

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
getrawpartition(void);
```

DESCRIPTION

getrawpartition() returns the partition number (‘a’ == 0, ‘b’ == 1, ...) of the “raw” partition of the system’s disks, or −1 in case of an error, setting the global *errno* variable. The possible values for *errno* are the same as in `sysctl`(3). The “raw” partition is defined as the partition which provides access to the entire disk, regardless of the disk’s partition map.

SEE ALSO

`getmaxpartitions`(3), `sysctl`(3)

HISTORY

The **getrawpartition** function call appeared in NetBSD 1.2.

NAME

getrpcent, **getrpcbyname**, **getrpcbynumber**, **endrpcent**, **setrpcent** — get RPC entry

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <rpc/rpc.h>

struct rpcent *
getrpcent(void);

struct rpcent *
getrpcbyname(const char *name);

struct rpcent *
getrpcbynumber(int number);

void
setrpcent(int stayopen);

void
endrpcent(void);
```

DESCRIPTION

getrpcent(), **getrpcbyname**(), and **getrpcbynumber**(), each return a pointer to an object with the following structure containing the broken-out fields of a line in the rpc program number data base, /etc/rpc:

```
struct rpcent {
        char    *r_name;        /* name of server for this rpc program */
        char    **r_aliases;    /* alias list */
        long    r_number;       /* rpc program number */
};
```

The members of this structure are:

r_name The name of the server for this rpc program.

r_aliases A zero terminated list of alternative names for the rpc program.

r_number The rpc program number for this service.

getrpcent() reads the next line of the file, opening the file if necessary.

setrpcent() opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getrpcent**() (either directly, or indirectly through one of the other "getrpc" calls).

endrpcent() closes the file.

getrpcbyname() and **getrpcbynumber**() sequentially search from the beginning of the file until a matching rpc program name or program number is found, or until end-of-file is encountered.

FILES

/etc/rpc

DIAGNOSTICS

A `NULL` pointer is returned on `EOF` or error.

SEE ALSO

`rpc`(5), `rpcinfo`(8), `ypserv`(8)

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getrpcport — get RPC port number

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

int
getrpcport(*char *host*, *int prognum*, *int versnum*, *int proto*);

DESCRIPTION

getrpcport() returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns 0 if it cannot contact the portmapper, or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it will still return a port number (for some version of the program) indicating that the program is indeed registered. The version mismatch will be detected upon the first call to the service.

NAME

getservent, **getservbyport**, **getservbyname**, **setservent**, **endservent** — get service entry

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <netdb.h>

struct servent *
getservent();

struct servent *
getservbyname(const char *name, const char *proto);

struct servent *
getservbyport(int port, const char *proto);

void
setservent(int stayopen);

void
endservent(void);
```

DESCRIPTION

The **getservent**(), **getservbyname**(), and **getservbyport**() functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, `/etc/services`.

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name The official name of the service.

s_aliases A NULL terminated list of alternative names for the service.

s_port The port number at which the service resides. Port numbers must be given and are returned in network byte order.

s_proto The name of the protocol to use when contacting the service.

The **getservent**() function reads the next line of the file, opening the file if necessary.

The **setservent**() function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getservbyname**() or **getservbyport**().

The **endservent**() function closes the file.

The **getservbyname**() and **getservbyport**() functions sequentially search from the beginning of the file until a matching protocol name or port number is found, or until `EOF` is encountered. If a protocol name is also supplied (non-`NULL`), searches must also match the protocol.

FILES

`/etc/services`

DIAGNOSTICS

Null pointer (0) returned on `EOF` or error.

SEE ALSO

`getprotoent`(3), `services`(5)

HISTORY

The **`getservent`**(), **`getservbyport`**(), **`getservbyname`**(), **`setservent`**(), and **`endservent`**() functions appeared in 4.2BSD.

BUGS

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Expecting port numbers to fit in a 32 bit quantity is probably naive.

NAME

getsubopt — get sub options from an argument

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

extern char *suboptarg

int
getsubopt(char **optionp, char * const *tokens, char **valuep);
```

DESCRIPTION

The **getsubopt**() function parses a string containing tokens delimited by one or more tab, space or comma (‘,’) characters. It is intended for use in parsing groups of option arguments provided as part of a utility command line.

The argument *optionp* is a pointer to a pointer to the string. The argument *tokens* is a pointer to a NULL-terminated array of pointers to strings.

The **getsubopt**() function returns the zero-based offset of the pointer in the *tokens* array referencing a string which matches the first token in the string, or −1 if the string contains no tokens or *tokens* does not contain a matching string.

If the token is of the form ‘‘name=value’’, the location referenced by *valuep* will be set to point to the start of the ‘‘value’’ portion of the token.

On return from **getsubopt**(), *optionp* will be set to point to the start of the next token in the string, or the null at the end of the string if no more tokens are present. The external variable *suboptarg* will be set to point to the start of the current token, or NULL if no tokens were present. The argument *valuep* will be set to point to the ‘‘value’’ portion of the token, or NULL if no ‘‘value’’ portion was present.

EXAMPLES

```
char *tokens[] = {
        #define ONE     0
                "one",
        #define TWO     1
                "two",
        NULL
};

...

extern char *optarg, *suboptarg;
char *options, *value;

while ((ch = getopt(argc, argv, "ab:")) != -1) {
        switch(ch) {
        case 'a':
                /* process ‘‘a’’ option */
                break;
        case 'b':
                options = optarg;
```

```
                while (*options) {
                        switch(getsubopt(&options, tokens, &value)) {
                        case ONE:
                                /* process ``one'' sub option */
                                break;
                        case TWO:
                                /* process ``two'' sub option */
                                if (!value)
                                        error("no value for two");
                                i = atoi(value);
                                break;
                        case -1:
                                if (suboptarg)
                                        error("unknown sub option %s",
                                          suboptarg);
                                else
                                        error("missing sub option");
                                break;
                }
                break;
        }
```

SEE ALSO

`getopt`(3), `strsep`(3)

HISTORY

The **`getsubopt`**() function first appeared in 4.4BSD.

NAME

gettext, dgettext, dcgettext − translate message

SYNOPSIS

#include <libintl.h>

char * gettext (const char * *msgid***);**
char * dgettext (const char * *domainname***, const char *** *msgid***);**
char * dcgettext (const char * *domainname***, const char *** *msgid***,**
int *category***);**

DESCRIPTION

The **gettext**, **dgettext** and **dcgettext** functions attempt to translate a text string into the user's native language, by looking up the translation in a message catalog.

The *msgid* argument identifies the message to be translated. By convention, it is the English version of the message, with non-ASCII characters replaced by ASCII approximations. This choice allows the translators to work with message catalogs, called PO files, that contain both the English and the translated versions of each message, and can be installed using the **msgfmt** utility.

A message domain is a set of translatable *msgid* messages. Usually, every software package has its own message domain. The domain name is used to determine the message catalog where the translation is looked up; it must be a non-empty string. For the **gettext** function, it is specified through a preceding **textdomain** call. For the **dgettext** and **dcgettext** functions, it is passed as the *domainname* argument; if this argument is NULL, the domain name specified through a preceding **textdomain** call is used instead.

Translation lookup operates in the context of the current locale. For the **gettext** and **dgettext** functions, the **LC_MESSAGES** locale facet is used. It is determined by a preceding call to the **setlocale** function. **setlocale(LC_ALL,"")** initializes the **LC_MESSAGES** locale based on the first nonempty value of the three environment variables **LC_ALL**, **LC_MESSAGES**, **LANG**; see **setlocale**(3). For the **dcgettext** function, the locale facet is determined by the *category* argument, which should be one of the **LC_xxx** constants defined in the <locale.h> header, excluding **LC_ALL**. In both cases, the functions also use the **LC_CTYPE** locale facet in order to convert the translated message from the translator's codeset to the current locale's codeset, unless overridden by a prior call to the **bind_textdomain_codeset** function.

The message catalog used by the functions is at the pathname *dirname*/*locale*/*category*/*domainname*.mo. Here *dirname* is the directory specified through **bindtextdomain**. Its default is system and configuration dependent; typically it is *prefix*/share/locale, where *prefix* is the installation prefix of the package. *locale* is the name of the current locale facet; the GNU implementation also tries generalizations, such as the language name without the territory name. *category* is **LC_MESSAGES** for the **gettext** and **dgettext** functions, or the argument passed to the **dcgettext** function.

If the **LANGUAGE** environment variable is set to a nonempty value, and the locale is not the "C" locale, the value of **LANGUAGE** is assumed to contain a colon separated list of locale names. The functions will attempt to look up a translation of *msgid* in each of the locales in turn. This is a GNU extension.

In the "C" locale, or if none of the used catalogs contain a translation for *msgid*, the **gettext**, **dgettext** and **dcgettext** functions return *msgid*.

RETURN VALUE

If a translation was found in one of the specified catalogs, it is converted to the locale's codeset and returned. The resulting string is statically allocated and must not be modified or freed. Otherwise *msgid* is returned.

ERRORS

errno is not modified.

BUGS

The return type ought to be **const char ***, but is **char *** to avoid warnings in C code predating ANSI C.

When an empty string is used for *msgid*, the functions may return a nonempty string.

SEE ALSO

ngettext(3), **dngettext**(3), **dcngettext**(3), **setlocale**(3), **textdomain**(3), **bindtextdomain**(3), **bind_textdomain_codeset**(3), **msgfmt**(1)

NAME

gettext, **dgettext**, **ngettext**, **dngettext**, **textdomain**, **bindtextdomain**, **bind_textdomain_codeset**, **dcgettext**, **dcngettext** — message handling functions

LIBRARY

Internationalized Message Handling Library (libintl, −lintl)

SYNOPSIS

```
#include <libintl.h>

char *
gettext(const char *msgid);

char *
dgettext(const char *domainname, const char *msgid);

char *
ngettext(const char *msgid1, const char *msgid2, unsigned long int n);

char *
dngettext(const char *domainname, const char *msgid1, const char *msgid2,
    unsigned long int n);

char *
textdomain(const char *domainname);

char *
bindtextdomain(const char *domainname, const char *dirname);

char *
bind_textdomain_codeset(const char *domainname, const char *codeset);

#include <libintl.h>
#include <locale.h>

char *
dcgettext(const char *domainname, const char *msgid, int category);

char *
dcngettext(const char *domainname, const char *msgid1, const char *msgid2,
    unsigned long int n, int category);
```

DESCRIPTION

The **gettext**(), **dgettext**(), and **dcgettext**() functions attempt to retrieve a target string based on the specified *msgid* argument within the context of a specific domain and the current locale. The length of strings returned by **gettext**(), **dgettext**(), and **dcgettext**() is undetermined until the function is called. The *msgid* argument is a nul-terminated string.

The **ngettext**(), **dngettext**(), and **dcngettext**() functions are equivalent to **gettext**(), **dgettext**(), and **dcgettext**(), respectively, except for the handling of plural forms. The **ngettext**(), **dngettext**(), and **dcngettext**() functions search for the message string using the *msgid1* argument as the key, using the argument *n* to determine the plural form. If no message catalogs are found, *msgid1* is returned if *n* == 1, otherwise *msgid2* is returned.

The LANGUAGE environment variable is examined first to determine the message catalogs to be used. The value of the LANGUAGE environment variable is a list of locale names separated by colon (:) character. If the LANGUAGE environment variable is defined, each locale name is tried in the specified order and if a message catalog containing the requested message is found, the message is returned. If the LANGUAGE environment

variable is defined but failed to locate a message catalog, the *msgid* string will be returned.

If the LANGUAGE environment variable is not defined, LC_ALL, LC_xxx, and LANG environment variables are examined to locate the message catalog, following the convention used by the setlocale(3) function.

The pathname used to locate the message catalog is dirname/locale/category/domainname.mo, where dirname is the directory specified by **bindtextdomain**(), locale is a locale name determined by the definition of environment variables, *category* is LC_MESSAGES if **gettext**(), **ngettext**(), **dgettext**(), or **dngettext**() is called, otherwise LC_xxx where the name is the same as the locale category name specified by the *category* argument of **dcgettext**() or **dcngettext**(). *domainname* is the name of the domain specified by **textdomain**() or the *domainname* argument of **dgettext**(), **dngettext**(), **dcgettext**(), or **dcngettext**().

For **gettext**() and **ngettext**(), the domain used is set by the last valid call to **textdomain**(). If a valid call to **textdomain**() has not been made, the default domain (called messages) is used.

For **dgettext**(), **dngettext**(), **dcgettext**(), and **dcngettext**(), the domain used is specified by the *domainname* argument. The *domainname* argument is equivalent in syntax and meaning to the *domainname* argument to **textdomain**(), except that the selection of the domain is valid only for the duration of the **dgettext**(), **dngettext**(), **dcgettext**(), or **dcngettext**() function call.

The **dcgettext**() and **dcngettext**() functions require additional argument *category* for retrieving message string for other than LC_MESSAGES category. Available value for the *category* argument are LC_CTYPE, LC_COLLATE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC, and LC_TIME. The call of **dcgettext**(*domainname*, *msgid*, *LC_MESSAGES*) is equivalent to **dgettext**(*domainname*, *msgid*). Note that LC_ALL must not be used.

The **textdomain**() function sets or queries the name of the current domain of the active LC_MESSAGES locale category. The *domainname* argument is a nul-terminated string that can contain only the characters allowed in legal filenames.

The *domainname* argument is the unique name of a domain on the system. If there are multiple versions of the same domain on one system, namespace collisions can be avoided by using **bindtextdomain**(). If **textdomain**() is not called, a default domain is selected. The setting of domain made by the last valid call to **textdomain**() remains valid across subsequent calls to setlocale(3), and **gettext**().

The *domainname* argument is applied to the currently active LC_MESSAGES locale.

The current setting of the domain can be queried without affecting the current state of the domain by calling **textdomain**() with *domainname* set to the NULL pointer. Calling **textdomain**() with a *domainname* argument of a NULL string sets the domain to the default domain (messages).

The **bindtextdomain**() function binds the path predicate for a message domain *domainname* to the value contained in dirname. If *domainname* is a non-empty string and has not been bound previously, **bindtextdomain**() binds *domainname* with *dirname*.

If *domainname* is a non-empty string and has been bound previously, **bindtextdomain**() replaces the old binding with dirname. The dirname argument can be an absolute pathname being resolved when **gettext**(), **ngettext**(), **dgettext**(), **dngettext**(), **dcgettext**(), or **dcngettext**() are called. If *domainname* is a NULL pointer or an empty string, **bindtextdomain**() returns a NULL pointer. If **bindtextdomain**() is not called, implementation-defined default directory is used.

The **bind_textdomain_codeset**() function can be used to specify the output *codeset* for message catalogs for domain *domainname*. The *codeset* argument must be a valid codeset name which can be used for the iconv_open(3) function.

If the *codeset* argument is the NULL pointer, **bind_textdomain_codeset**() returns the currently selected *codeset* for the domain with the name *domainname*. It returns a NULL pointer if no *codeset*

has yet been selected.

The **bind_textdomain_codeset**() function can be used several times. If used multiple times, with the same *domainname* argument, the later call overrides the settings made by the earlier one.

The **bind_textdomain_codeset**() function returns a pointer to a string containing the name of the selected *codeset*.

SEE ALSO

setlocale(3), nls(7)

HISTORY

The functions are implemented by Citrus project, based on the documentations for GNU gettext.

BUGS

bind_textdomain_codeset() does not work at this moment (it always fails).

NAME

getttyent, **getttynam**, **setttyent**, **setttyentpath**, **endttyent** — get ttys file entry

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ttyent.h>

struct ttyent *
getttyent();

struct ttyent *
getttynam(char *name);

int
setttyent(void);

int
setttyentpath(const char *path);

int
endttyent(void);
```

DESCRIPTION

The **getttyent**(), and **getttynam**() functions each return a pointer to an object, with the following structure, containing the broken-out fields of a line from the tty description file.

```
struct ttyent {
        char    *ty_name;       /* terminal device name */
        char    *ty_getty;      /* command to execute */
        char    *ty_type;       /* terminal type */
#define TTY_ON          0x01    /* enable logins */
#define TTY_SECURE      0x02    /* allow uid of 0 to login */
#define TTY_LOCAL       0x04    /* set 'CLOCAL' on open (dev. specific) */
#define TTY_RTSCTS      0x08    /* set 'CRTSCTS' on open (dev. specific) */
#define TTY_SOFTCAR     0x10    /* ignore hardware carrier (dev. spec.) */
#define TTY_MDMBUF      0x20    /* set 'MDMBUF' on open (dev. specific) */
#define TTY_DTRCTS      0x40    /* set 'CDTRCTS' on open (dev. specific) */
        int     ty_status;      /* flag values */
        char    *ty_window;     /* command for window manager */
        char    *ty_comment;    /* comment field */
        char    *ty_class;      /* category of tty usage */
};
```

The fields are as follows:

ty_name The name of the character-special file.

ty_getty The name of the command invoked by `init`(8) to initialize tty line characteristics.

ty_type The name of the default terminal type connected to this tty line.

ty_status A mask of bit fields which indicate various actions allowed on this tty line. The possible flags are as follows:

TTY_ON — Enables logins (i.e., `init`(8) will start the command referenced by *ty_getty* on this entry).

TTY_SECURE — Allow users with a uid of 0 to login on this terminal.

TTY_LOCAL — If the terminal port's driver supports it, cause the line to be treated as ''local.''

TTY_MDMBUF — If the terminal port's driver supports it, use DTR/DCD hardware flow control on the line by default.

TTY_RTSCTS — If the terminal port's driver supports it, use full-duplex RTS/CTS hardware flow control on the line by default.

TTY_SOFTCAR — If the terminal port's driver supports it, ignore hardware carrier on the line.

ty_window — The command to execute for a window system associated with the line.

ty_comment — Any trailing comment field, with any leading hash marks (''#'') or whitespace removed.

ty_class — A key indexing into a termcap-style database (/etc/ttyclasses) of attributes for this class of tty. No attributes are currently defined or used, so there are currently no functions to retrieve them.

If any of the fields pointing to character strings are unspecified, they are returned as null pointers. The field *ty_status* will be zero if no flag values are specified.

See `ttys`(5) for a more complete discussion of the meaning and usage of the fields.

The **getttyent**() function reads the next line from the ttys file, opening the file if necessary. The **setttyent**() function rewinds the file if open, or opens the file if it is unopened. The **setttyentpath**() function is equivalent to **setttyent**() but accepts an additional argument to read the ttys information from an alternate file instead of the default location (defined in `_PATH_TTYS`). The **endttyent**() function closes any open files.

The **getttynam**() function searches from the beginning of the file until a matching *name* is found (or until `EOF` is encountered).

RETURN VALUES

The routines **getttyent**() and **getttynam**() return a null pointer on `EOF` or error. The **setttyent**() and **setttyentpath**() functions and **endttyent**() return 0 on failure and 1 on success.

FILES

`/etc/ttys`

SEE ALSO

`login`(1), `ttyslot`(3), `gettytab`(5), `termcap`(5), `ttys`(5), `getty`(8), `init`(8), `ttyflags`(8)

HISTORY

The **getttyent**(), **getttynam**(), **setttyent**(), and **endttyent**() functions appeared in 4.3BSD. The **setttyentpath**() function appeared in NetBSD 4.0.

BUGS

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it.

NAME

getusershell, **setusershell**, **endusershell** — get valid user shells

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

char *
getusershell(void);

void
setusershell(void);

void
endusershell(void);
```

DESCRIPTION

The **getusershell**() function returns a pointer to a valid user shell as defined by the system manager in the shells database as described in shells(5). If the shells database is not available, **getusershell**() behaves as if /bin/sh and /bin/csh were listed.

The **getusershell**() function reads the next line (opening the file if necessary); **setusershell**() rewinds the file; **endusershell**() closes it.

FILES

/etc/shells

DIAGNOSTICS

The routine **getusershell**() returns a null pointer (0) on EOF.

SEE ALSO

nsswitch.conf(5), shells(5)

HISTORY

The **getusershell**() function appeared in 4.3BSD.

BUGS

The **getusershell**() function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to **getusershell**() will modify the same object.

NAME

fgetwc, **getwc**, **getwchar**, — get next wide-character from input stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t
fgetwc(FILE *stream);

wint_t
getwc(FILE *stream);

wint_t
getwchar();
```

DESCRIPTION

The **fgetwc**() function obtains the next input wide-character (if present) from the stream pointed at by *stream*, or the next character pushed back on the stream via ungetwc(3).

The **getwc**() function acts essentially identically to **fgetwc**(), but is a macro that expands in-line.

The **getwchar**() function is equivalent to **getwc**() with the argument stdin.

RETURN VALUES

If successful, these routines return the next wide-character from the *stream*. If the stream is at end-of-file or a read error occurs, the routines return WEOF. The routines feof(3) and ferror(3) must be used to distinguish between end-of-file and error. If an error occurs, the global variable *errno* is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return WEOF until the condition is cleared with clearerr(3).

SEE ALSO

ferror(3), fopen(3), fread(3), putwc(3), stdio(3), ungetwc(3)

STANDARDS

The **fgetwc**(), **getwc**() and **getwchar**() functions conform to ISO/IEC 9899:1999 (“ISO C99”).

NAME

glob, **globfree** — generate pathnames matching a pattern

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <glob.h>

int
glob(const char * restrict pattern, int flags,
    const int (*errfunc)(const char *, int), glob_t * restrict pglob);

void
globfree(glob_t *pglob);
```

DESCRIPTION

The **glob**() function is a pathname generator that implements the rules for file name pattern matching used by the shell.

The include file `glob.h` defines the structure type *glob_t*, which contains at least the following fields:

```
typedef struct {
        size_t gl_pathc;        /* count of total paths so far */
        size_t gl_matchc;       /* count of paths matching pattern */
        size_t gl_offs;                 /* reserved at beginning of gl_pathv */
        int gl_flags;           /* returned flags */
        char **gl_pathv;        /* list of paths matching pattern */
} glob_t;
```

The argument *pattern* is a pointer to a pathname pattern to be expanded. The **glob**() argument matches all accessible pathnames against the pattern and creates a list of the pathnames that match. In order to have access to a pathname, **glob**() requires search permission on every component of a path except the last and read permission on each directory of any filename component of *pattern* that contains any of the special characters '*', '?' or '['.

The **glob**() argument stores the number of matched pathnames into the *gl_pathc* field, and a pointer to a list of pointers to pathnames into the *gl_pathv* field. The first pointer after the last pathname is `NULL`. If the pattern does not match any pathnames, the returned number of matched paths is set to zero.

It is the caller's responsibility to create the structure pointed to by *pglob*. The **glob**() function allocates other space as needed, including the memory pointed to by *gl_pathv*.

The argument *flags* is used to modify the behavior of **glob**(). The value of *flags* is the bitwise inclusive OR of any of the following values defined in `glob.h`:

`GLOB_APPEND` Append pathnames generated to the ones from a previous call (or calls) to **glob**(). The value of *gl_pathc* will be the total matches found by this call and the previous call(s). The pathnames are appended to, not merged with the pathnames returned by the previous call(s). Between calls, the caller must not change the setting of the `GLOB_DOOFFS` flag, nor change the value of *gl_offs* when `GLOB_DOOFFS` is set, nor (obviously) call **globfree**() for *pglob*.

`GLOB_DOOFFS` Make use of the *gl_offs* field. If this flag is set, *gl_offs* is used to specify how many `NULL` pointers to prepend to the beginning of the *gl_pathv* field. In other words, *gl_pathv* will point to *gl_offs* `NULL` pointers, followed by *gl_pathc* pathname pointers, followed by a `NULL` pointer.

GLOB_ERR Causes **glob**() to return when it encounters a directory that it cannot open or read. Ordinarily, **glob**() continues to find matches.

GLOB_MARK Each pathname that is a directory that matches *pattern* has a slash appended.

GLOB_NOCHECK If *pattern* does not match any pathname, then **glob**() returns a list consisting of only *pattern*, with the number of total pathnames set to 1, and the number of matched pathnames set to 0.

GLOB_NOSORT By default, the pathnames are sorted in ascending ASCII order; this flag prevents that sorting (speeding up **glob**()).

The following values may also be included in *flags*, however, they are non-standard extensions to IEEE Std 1003.2 ("POSIX.2").

GLOB_ALTDIRFUNC The following additional fields in the pglob structure have been initialized with alternate functions for glob to use to open, read, and close directories and to get stat information on names found in those directories.

```
void *(*gl_opendir)(const char * name);
struct dirent *(*gl_readdir)(void *);
void (*gl_closedir)(void *);
int (*gl_lstat)(const char *name, struct stat *st);
int (*gl_stat)(const char *name, struct stat *st);
```

This extension is provided to allow programs such as restore(8) to provide globbing from directories stored on tape.

GLOB_BRACE Pre-process the pattern string to expand {pat,pat,...} strings like csh(1). The pattern '{}' is left unexpanded for historical reasons (csh(1) does the same thing to ease typing of find(1) patterns).

GLOB_MAGCHAR Set by the **glob**() function if the pattern included globbing characters. See the description of the usage of the *gl_matchc* structure member for more details.

GLOB_NOMAGIC Is the same as GLOB_NOCHECK but it only appends the *pattern* if it does not contain any of the special characters "*", "?" or "[". GLOB_NOMAGIC is provided to simplify implementing the historic csh(1) globbing behavior and should probably not be used anywhere else.

GLOB_NOESCAPE Disable the use of the backslash ('\') character for quoting.

GLOB_TILDE Expand patterns that start with '~' to user name home directories.

GLOB_LIMIT Limit the amount of memory used by matches to ARG_MAX. This option should be set for programs that can be coerced to a denial of service attack via patterns that expand to a very large number of matches, such as a long string of */../*/..

GLOB_PERIOD Allow metacharacters to match a leading period in a filename.

GLOB_NO_DOTDIRS Hide '.' and '..' from metacharacter matches, regardless of whether GLOB_PERIOD is set and whether the pattern component begins with a literal period.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is non-NULL, **glob**() calls *(*errfunc)(path, errno)*. This may be unintuitive: a pattern like */Makefile will try to stat(2) foo/Makefile even if foo is not a directory, resulting in a call to *errfunc*. The error routine can suppress this action by testing for ENOENT and ENOTDIR; however, the GLOB_ERR flag will still cause an immediate return when this happens.

If *errfunc* returns non-zero, **glob**() stops the scan and returns GLOB_ABORTED after setting *gl_pathc* and *gl_pathv* to reflect any paths already matched. This also happens if an error is encountered and GLOB_ERR is set in *flags*, regardless of the return value of *errfunc*, if called. If GLOB_ERR is not set and either *errfunc* is NULL or *errfunc* returns zero, the error is ignored.

The **globfree**() function frees any space associated with *pglob* from a previous call(s) to **glob**().

The historical GLOB_QUOTE flag is no longer supported. Per IEEE Std 1003.2-1992 ("POSIX.2"), backslash escaping of special characters is the default behaviour; it may be disabled by specifying the GLOB_NOESCAPE flag.

RETURN VALUES

On successful completion, **glob**() returns zero. In addition the fields of *pglob* contain the values described below:

gl_pathc contains the total number of matched pathnames so far. This includes other matches from previous invocations of **glob**() if GLOB_APPEND was specified.

gl_matchc contains the number of matched pathnames in the current invocation of **glob**().

gl_flags contains a copy of the *flags* parameter with the bit GLOB_MAGCHAR set if *pattern* contained any of the special characters "*", "?" or "[", cleared if not.

gl_pathv contains a pointer to a NULL-terminated list of matched pathnames. However, if *gl_pathc* is zero, the contents of *gl_pathv* are undefined.

If **glob**() terminates due to an error, it sets *errno* and returns one of the following non-zero constants, which are defined in the include file ⟨glob.h⟩:

GLOB_ABORTED The scan was stopped because an error was encountered and either GLOB_ERR was set or *(*errfunc)()* returned non-zero.

GLOB_NOMATCH The pattern does not match any existing pathname, and GLOB_NOCHECK was not set in flags.

GLOB_NOSPACE An attempt to allocate memory failed, or if *errno* was 0 GLOB_LIMIT was specified in the flags and ARG_MAX patterns were matched.

The historical GLOB_ABEND return constant is no longer supported. Portable applications should use the GLOB_ABORTED constant instead.

The arguments *pglob->gl_pathc* and *pglob->gl_pathv* are still set as specified above.

ENVIRONMENT

HOME If defined, used as the home directory of the current user in tilde expansions.

EXAMPLES

A rough equivalent of `ls -l *.c *.h` can be obtained with the following code:

```
glob_t g;

g.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &g);
glob("*.h", GLOB_DOOFFS | GLOB_APPEND, NULL, &g);
g.gl_pathv[0] = "ls";
g.gl_pathv[1] = "-l";
execvp("ls", g.gl_pathv);
```

SEE ALSO

sh(1), fnmatch(3), regexp(3)

STANDARDS

The **glob**() function is expected to be IEEE Std 1003.2 ("POSIX.2") compatible with the exception that the flags GLOB_ALTDIRFUNC, GLOB_BRACE, GLOB_MAGCHAR, GLOB_NOMAGIC, GLOB_TILDE, and GLOB_LIMIT and the fields *gl_matchc* and *gl_flags* should not be used by applications striving for strict POSIX conformance.

HISTORY

The **glob**() and **globfree**() functions first appeared in 4.4BSD.

BUGS

Patterns longer than MAXPATHLEN may cause unchecked errors.

The **glob**() function may fail and set *errno* for any of the errors specified for the library routines stat(2), closedir(3), opendir(3), readdir(3), malloc(3), and free(3).

NAME

grantpt — grant access to a slave pseudo-terminal device

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
grantpt(int fildes);
```

DESCRIPTION

The **grantpt**() function changes the mode and ownership of the slave pseudo-terminal device that corresponds to the master pseudo-terminal device associated with *fildes* to be owned by the real user id of the calling process, group id of `tty`. The permissions are set to readable and writable by owner, and writable by group. If the slave pseudo-terminal device was being accessed by other file descriptors at the time, all such access will be revoked.

RETURN VALUES

If successful, **grantpt**() returns 0; otherwise a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

The **grantpt**() function will fail if:

`[EACCESS]` the corresponding pseudo-terminal device could not be accessed.

`[EBADF]` *fildes* is not a valid descriptor.

`[EINVAL]` *fildes* is not associated with a master pseudo-terminal device.

NOTES

Setting the group to `tty` and revoking accesses by other file descriptors are NetBSD extensions. Calling **grantpt**() is equivalent to:

```
ioctl(fildes, TIOCGRANTPT, 0);
```

SEE ALSO

`ioctl`(2), `posix_openpt`(3), `ptsname`(3), `unlockpt`(3)

STANDARDS

The **grantpt**() function conforms to IEEE Std 1003.1-2001 (“POSIX.1”). Its first release was in X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”).

NAME

gss_accept_sec_context, **gss_acquire_cred**, **gss_add_cred**, **gss_add_oid_set_member**, **gss_canonicalize_name**, **gss_compare_name**, **gss_context_time**, **gss_create_empty_oid_set**, **gss_delete_sec_context**, **gss_display_name**, **gss_display_status**, **gss_duplicate_name**, **gss_export_name**, **gss_export_sec_context**, **gss_get_mic**, **gss_import_name**, **gss_import_sec_context**, **gss_indicate_mechs**, **gss_init_sec_context**, **gss_inquire_context**, **gss_inquire_cred**, **gss_inquire_cred_by_mech**, **gss_inquire_mechs_for_name**, **gss_inquire_names_for_mech**, **gss_krb5_ccache_name**, **gss_krb5_compat_des3_mic**, **gss_krb5_copy_ccache**, **gss_krb5_import_cred** **gsskrb5_extract_authz_data_from_sec_context**, **gsskrb5_register_acceptor_identity**, **gss_krb5_import_ccache**, **gss_krb5_get_tkt_flags**, **gss_process_context_token**, **gss_release_buffer**, **gss_release_cred**, **gss_release_name**, **gss_release_oid_set**, **gss_seal**, **gss_sign**, **gss_test_oid_set_member**, **gss_unseal**, **gss_unwrap**, **gss_verify**, **gss_verify_mic**, **gss_wrap**, **gss_wrap_size_limit** — Generic Security Service Application Program Interface library

LIBRARY

GSS-API library (libgssapi, -lgssapi)

SYNOPSIS

```
#include <gssapi/gssapi.h>

OM_uint32
gss_accept_sec_context(OM_uint32 * minor_status,
    gss_ctx_id_t * context_handle,
    const gss_cred_id_t acceptor_cred_handle,
    const gss_buffer_t input_token_buffer,
    const gss_channel_bindings_t input_chan_bindings,
    gss_name_t * src_name,gss_OID * mech_type,gss_buffer_t output_token,
    OM_uint32 * ret_flags,OM_uint32 * time_rec,
    gss_cred_id_t * delegated_cred_handle);

OM_uint32
gss_acquire_cred(OM_uint32 * minor_status,const gss_name_t desired_name,
    OM_uint32 time_req,const gss_OID_set desired_mechs,
    gss_cred_usage_t cred_usage,gss_cred_id_t * output_cred_handle,
    gss_OID_set * actual_mechs,OM_uint32 * time_rec);

OM_uint32
gss_add_cred(OM_uint32 *minor_status,
    const gss_cred_id_t input_cred_handle,const gss_name_t desired_name,
    const gss_OID desired_mech,gss_cred_usage_t cred_usage,
    OM_uint32 initiator_time_req,OM_uint32 acceptor_time_req,
    gss_cred_id_t *output_cred_handle,gss_OID_set *actual_mechs,
    OM_uint32 *initiator_time_rec,OM_uint32 *acceptor_time_rec);

OM_uint32
gss_add_oid_set_member(OM_uint32 * minor_status,const gss_OID member_oid,
    gss_OID_set * oid_set);

OM_uint32
gss_canonicalize_name(OM_uint32 * minor_status,
    const gss_name_t input_name,const gss_OID mech_type,
```

*gss_name_t * output_name*);

OM_uint32
gss_compare_name(*OM_uint32 * minor_status*,*const gss_name_t name1*,
const gss_name_t name2,*int * name_equal*);

OM_uint32
gss_context_time(*OM_uint32 * minor_status*,
const gss_ctx_id_t context_handle,*OM_uint32 * time_rec*);

OM_uint32
gss_create_empty_oid_set(*OM_uint32 * minor_status*,*gss_OID_set * oid_set*);

OM_uint32
gss_delete_sec_context(*OM_uint32 * minor_status*,
*gss_ctx_id_t * context_handle*,*gss_buffer_t output_token*);

OM_uint32
gss_display_name(*OM_uint32 * minor_status*,*const gss_name_t input_name*,
gss_buffer_t output_name_buffer,*gss_OID * output_name_type*);

OM_uint32
gss_display_status(*OM_uint32 *minor_status*,*OM_uint32 status_value*,
int status_type,*const gss_OID mech_type*,*OM_uint32 *message_context*,
gss_buffer_t status_string);

OM_uint32
gss_duplicate_name(*OM_uint32 * minor_status*,*const gss_name_t src_name*,
*gss_name_t * dest_name*);

OM_uint32
gss_export_name(*OM_uint32 * minor_status*,*const gss_name_t input_name*,
gss_buffer_t exported_name);

OM_uint32
gss_export_sec_context(*OM_uint32 * minor_status*,
*gss_ctx_id_t * context_handle*,*gss_buffer_t interprocess_token*);

OM_uint32
gss_get_mic(*OM_uint32 * minor_status*,*const gss_ctx_id_t context_handle*,
gss_qop_t qop_req,*const gss_buffer_t message_buffer*,
gss_buffer_t message_token);

OM_uint32
gss_import_name(*OM_uint32 * minor_status*,
const gss_buffer_t input_name_buffer,*const gss_OID input_name_type*,
*gss_name_t * output_name*);

OM_uint32
gss_import_sec_context(*OM_uint32 * minor_status*,
const gss_buffer_t interprocess_token,*gss_ctx_id_t * context_handle*);

OM_uint32
gss_indicate_mechs(*OM_uint32 * minor_status*,*gss_OID_set * mech_set*);

OM_uint32
gss_init_sec_context(*OM_uint32 * minor_status*,
const gss_cred_id_t initiator_cred_handle,
*gss_ctx_id_t * context_handle*,*const gss_name_t target_name*,

const gss_OID mech_type, *OM_uint32 req_flags*, *OM_uint32 time_req*,
const gss_channel_bindings_t input_chan_bindings,
const gss_buffer_t input_token, *gss_OID * actual_mech_type*,
gss_buffer_t output_token, *OM_uint32 * ret_flags*,
*OM_uint32 * time_rec*);

OM_uint32
gss_inquire_context(*OM_uint32 * minor_status*,
const gss_ctx_id_t context_handle, *gss_name_t * src_name*,
*gss_name_t * targ_name*, *OM_uint32 * lifetime_rec*, *gss_OID * mech_type*,
*OM_uint32 * ctx_flags*, *int * locally_initiated*, *int * open_context*);

OM_uint32
gss_inquire_cred(*OM_uint32 * minor_status*,
const gss_cred_id_t cred_handle, *gss_name_t * name*,
*OM_uint32 * lifetime*, *gss_cred_usage_t * cred_usage*,
*gss_OID_set * mechanisms*);

OM_uint32
gss_inquire_cred_by_mech(*OM_uint32 * minor_status*,
const gss_cred_id_t cred_handle, *const gss_OID mech_type*,
*gss_name_t * name*, *OM_uint32 * initiator_lifetime*,
*OM_uint32 * acceptor_lifetime*, *gss_cred_usage_t * cred_usage*);

OM_uint32
gss_inquire_mechs_for_name(*OM_uint32 * minor_status*,
const gss_name_t input_name, *gss_OID_set * mech_types*);

OM_uint32
gss_inquire_names_for_mech(*OM_uint32 * minor_status*,
const gss_OID mechanism, *gss_OID_set * name_types*);

OM_uint32
gss_krb5_ccache_name(*OM_uint32 *minor*, *const char *name*,
*const char **old_name*);

OM_uint32
gss_krb5_copy_ccache(*OM_uint32 *minor*, *gss_cred_id_t cred*, *krb5_ccache out*);

OM_uint32
gss_krb5_import_cred(*OM_uint32 *minor_status*, *krb5_ccache id*,
krb5_principal keytab_principal, *krb5_keytab keytab*,
*gss_cred_id_t *cred*);

OM_uint32
gss_krb5_compat_des3_mic(*OM_uint32 * minor_status*,
gss_ctx_id_t context_handle, *int onoff*);

OM_uint32
gsskrb5_extract_authz_data_from_sec_context(*OM_uint32 *minor_status*,
gss_ctx_id_t context_handle, *int ad_type*, *gss_buffer_t ad_data*);

OM_uint32
gsskrb5_register_acceptor_identity(*const char *identity*);

OM_uint32
gss_krb5_import_cache(*OM_uint32 *minor*, *krb5_ccache id*, *krb5_keytab keytab*,
*gss_cred_id_t *cred*);

```
OM_uint32
gss_krb5_get_tkt_flags(OM_uint32 *minor_status,
    gss_ctx_id_t context_handle,OM_uint32 *tkt_flags);

OM_uint32
gss_process_context_token(OM_uint32 * minor_status,
    const gss_ctx_id_t context_handle,const gss_buffer_t token_buffer);

OM_uint32
gss_release_buffer(OM_uint32 * minor_status,gss_buffer_t buffer);

OM_uint32
gss_release_cred(OM_uint32 * minor_status,gss_cred_id_t * cred_handle);

OM_uint32
gss_release_name(OM_uint32 * minor_status,gss_name_t * input_name);

OM_uint32
gss_release_oid_set(OM_uint32 * minor_status,gss_OID_set * set);

OM_uint32
gss_seal(OM_uint32 * minor_status,gss_ctx_id_t context_handle,
    int conf_req_flag,int qop_req,gss_buffer_t input_message_buffer,
    int * conf_state,gss_buffer_t output_message_buffer);

OM_uint32
gss_sign(OM_uint32 * minor_status,gss_ctx_id_t context_handle,
    int qop_req,gss_buffer_t message_buffer,gss_buffer_t message_token);

OM_uint32
gss_test_oid_set_member(OM_uint32 * minor_status,const gss_OID member,
    const gss_OID_set set,int * present);

OM_uint32
gss_unseal(OM_uint32 * minor_status,gss_ctx_id_t context_handle,
    gss_buffer_t input_message_buffer,gss_buffer_t output_message_buffer,
    int * conf_state,int * qop_state);

OM_uint32
gss_unwrap(OM_uint32 * minor_status,const gss_ctx_id_t context_handle,
    const gss_buffer_t input_message_buffer,
    gss_buffer_t output_message_buffer,int * conf_state,
    gss_qop_t * qop_state);

OM_uint32
gss_verify(OM_uint32 * minor_status,gss_ctx_id_t context_handle,
    gss_buffer_t message_buffer,gss_buffer_t token_buffer,
    int * qop_state);

OM_uint32
gss_verify_mic(OM_uint32 * minor_status,
    const gss_ctx_id_t context_handle,const gss_buffer_t message_buffer,
    const gss_buffer_t token_buffer,gss_qop_t * qop_state);

OM_uint32
gss_wrap(OM_uint32 * minor_status,const gss_ctx_id_t context_handle,
    int conf_req_flag,gss_qop_t qop_req,
    const gss_buffer_t input_message_buffer,int * conf_state,
```

```
    gss_buffer_t output_message_buffer);

OM_uint32
gss_wrap_size_limit(OM_uint32 * minor_status,
    const gss_ctx_id_t context_handle,int conf_req_flag,
    gss_qop_t qop_req,OM_uint32 req_output_size,
    OM_uint32 * max_input_size);
```

DESCRIPTION

Generic Security Service API (GSS-API) version 2, and its C binding, is described in RFC2743 and RFC2744. Version 1 (deprecated) of the C binding is described in RFC1509.

Heimdals GSS-API implementation supports the following mechanisms

- GSS_KRB5_MECHANISM
- GSS_SPNEGO_MECHANISM

GSS-API have generic name types that all mechanism are supposed to implement (if possible):

- GSS_C_NT_USER_NAME
- GSS_C_NT_MACHINE_UID_NAME
- GSS_C_NT_STRING_UID_NAME
- GSS_C_NT_HOSTBASED_SERVICE
- GSS_C_NT_ANONYMOUS
- GSS_C_NT_EXPORT_NAME

GSS-API implementations that supports Kerberos 5 have some additional name types:

- GSS_KRB5_NT_PRINCIPAL_NAME
- GSS_KRB5_NT_USER_NAME
- GSS_KRB5_NT_MACHINE_UID_NAME
- GSS_KRB5_NT_STRING_UID_NAME

In GSS-API, names have two forms, internal names and contiguous string names.

- Internal name and mechanism name

  Internal names are implementation specific representation of a GSS-API name. Mechanism names special form of internal names corresponds to one and only one mechanism.

  In GSS-API an internal name is stored in a gss_name_t.

- Contiguous string name and exported name

  Contiguous string names are gssapi names stored in a OCTET STRING that together with a name type identifier (OID) uniquely specifies a gss-name. A special form of the contiguous string name is the exported name that have a OID embedded in the string to make it unique. Exported name have the name-type GSS_C_NT_EXPORT_NAME.

  In GSS-API an contiguous string name is stored in a gss_buffer_t.

  Exported names also have the property that they are specified by the mechanism itself and compatible between diffrent GSS-API implementations.

ACCESS CONTROL

There are two ways of comparing GSS-API names, either comparing two internal names with each other or two contiguous string names with either other.

To compare two internal names with each other, import (if needed) the names with **gss_import_name**() into the GSS-API implementation and the compare the imported name with **gss_compare_name**().

Importing names can be slow, so when its possible to store exported names in the access control list, comparing contiguous string name might be better.

when comparing contiguous string name, first export them into a GSS_C_NT_EXPORT_NAME name with **gss_export_name**() and then compare with memcmp(3).

Note that there are might be a difference between the two methods of comparing names. The first (using **gss_compare_name**()) will compare to (unauthenticated) names are the same. The second will compare if a mechanism will authenticate them as the same principal.

For example, if **gss_import_name**() name was used with GSS_C_NO_OID the default syntax is used for all mechanism the GSS-API implementation supports. When compare the imported name of GSS_C_NO_OID it may match serveral mechanism names (MN).

The resulting name from **gss_display_name**() must not be used for acccess control.

FUNCTIONS

gss_display_name() takes the gss name in *input_name* and puts a printable form in *output_name_buffer*. *output_name_buffer* should be freed when done using **gss_release_buffer**(). *output_name_type* can either be NULL or a pointer to a gss_OID and will in the latter case contain the OID type of the name. The name must only be used for printing. If access control is needed, see section **ACCESS CONTROL**.

gss_inquire_context() returns information about the context. Information is available even after the context have expired. *lifetime_rec* argument is set to GSS_C_INDEFINITE (dont expire) or the number of seconds that the context is still valid. A value of 0 means that the context is expired. *mech_type* argument should be considered readonly and must not be released. *src_name* and **dest_name**() are both mechanims names and must be released with **gss_release_name**() when no longer used.

gss_context_time will return the amount of time (in seconds) of the context is still valid. If its expired *time_rec* will be set to 0 and GSS_S_CONTEXT_EXPIRED returned.

gss_sign(), **gss_verify**(), **gss_seal**(), and **gss_unseal**() are part of the GSS-API V1 interface and are obsolete. The functions should not be used for new applications. They are provided so that version 1 applications can link against the library.

EXTENSIONS

gss_krb5_ccache_name() sets the internal kerberos 5 credential cache name to *name*. The old name is returned in *old_name*, and must not be freed. The data allocated for *old_name* is free upon next call to **gss_krb5_ccache_name**(). This function is not threadsafe if *old_name* argument is used.

gss_krb5_copy_ccache() will extract the krb5 credentials that are transferred from the initiator to the acceptor when using token delegation in the Kerberos mechanism. The acceptor receives the delegated token in the last argument to **gss_accept_sec_context**().

gss_krb5_import_cred() will import the krb5 credentials (both keytab and/or credential cache) into gss credential so it can be used withing GSS-API. The *ccache* is copied by reference and thus shared, so if the credential is destroyed with *krb5_cc_destroy*, all users of thep *gss_cred_id_t* returned by **gss_krb5_import_ccache**() will fail.

gsskrb5_register_acceptor_identity() sets the Kerberos 5 filebased keytab that the acceptor will use. The *identifier* is the file name.

gsskrb5_extract_authz_data_from_sec_context() extracts the Kerberos authorizationdata that may be stored within the context. Tha caller must free the returned buffer *ad_data* with **gss_release_buffer**() upon success.

gss_krb5_get_tkt_flags() return the ticket flags for the kerberos ticket receive when authenticating the initiator. Only valid on the acceptor context.

gss_krb5_compat_des3_mic() turns on or off the compatibility with older version of Heimdal using des3 get and verify mic, this is way to programmatically set the [gssapi]broken_des3_mic and [gssapi]correct_des3_mic flags (see COMPATIBILITY section in gssapi(3)). If the CPP symbol GSS_C_KRB5_COMPAT_DES3_MIC is present, **gss_krb5_compat_des3_mic**() exists. **gss_krb5_compat_des3_mic**() will be removed in a later version of the GSS-API library.

SEE ALSO

gssapi(3), krb5(3), krb5_ccache(3), kerberos(8)

NAME

gssapi — Generic Security Service Application Program Interface library

LIBRARY

GSS-API Library (libgssapi, -lgssapi)

DESCRIPTION

The Generic Security Service Application Program Interface (GSS-API) provides security services to callers in a generic fashion, supportable with a range of underlying mechanisms and technologies and hence allowing source-level portability of applications to different environments.

The GSS-API implementation in Heimdal implements the Kerberos 5 and the SPNEGO GSS-API security mechanisms.

LIST OF FUNCTIONS

These functions constitute the gssapi library, *libgssapi*. Declarations for these functions may be obtained from the include file `gssapi/gssapi.h`.

Name/Page *Description*

gss_accept_sec_context.3
gss_acquire_cred.3
gss_add_cred.3
gss_add_oid_set_member.3
gss_canonicalize_name.3
gss_compare_name.3
gss_context_time.3
gss_create_empty_oid_set.3
gss_delete_sec_context.3
gss_display_name.3
gss_display_status.3
gss_duplicate_name.3
gss_export_name.3
gss_export_sec_context.3
gss_get_mic.3
gss_import_name.3
gss_import_sec_context.3
gss_indicate_mechs.3
gss_init_sec_context.3
gss_inquire_context.3
gss_inquire_cred.3
gss_inquire_cred_by_mech.3
gss_inquire_mechs_for_name.3
gss_inquire_names_for_mech.3
gss_krb5_ccache_name.3
gss_krb5_compat_des3_mic.3
gss_krb5_copy_ccache.3
gss_krb5_extract_authz_data_from_sec_context.3
gss_krb5_import_ccache.3
gss_process_context_token.3
gss_release_buffer.3
gss_release_cred.3

gss_release_name.3
gss_release_oid_set.3
gss_seal.3
gss_sign.3
gss_test_oid_set_member.3
gss_unseal.3
gss_unwrap.3
gss_verify.3
gss_verify_mic.3
gss_wrap.3
gss_wrap_size_limit.3

COMPATIBILITY

The **Heimdal** GSS-API implementation had a bug in releases before 0.6 that made it fail to inter-operate when using DES3 with other GSS-API implementations when using **gss_get_mic**() / **gss_verify_mic**(). It is possible to modify the behavior of the generator of the MIC with the krb5.conf configuration file so that old clients/servers will still work.

New clients/servers will try both the old and new MIC in Heimdal 0.6. In 0.7 it will check only if configured - the compatibility code will be removed in 0.8.

Heimdal 0.6 still generates by default the broken GSS-API DES3 mic, this will change in 0.7 to generate correct des3 mic.

To turn on compatibility with older clients and servers, change the **[gssapi]** *broken_des3_mic* in krb5.conf that contains a list of globbing expressions that will be matched against the server name. To turn off generation of the old (incompatible) mic of the MIC use **[gssapi]** *correct_des3_mic*.

If a match for a entry is in both **[gssapi]** *correct_des3_mic* and **[gssapi]** *broken_des3_mic*, the later will override.

This config option modifies behaviour for both clients and servers.

Microsoft implemented SPNEGO to Windows2000, however, they manage to get it wrong, their implementation didn't fill in the MechListMIC in the reply token with the right content. There is a work around for this problem, but not all implementation support it.

Heimdal defaults to correct SPNEGO when the the kerberos implementation uses CFX, or when it is configured by the user. To turn on compatibility with peers, use option **[gssapi]** *require_mechlist_mic*.

EXAMPLES

```
[gssapi]
        broken_des3_mic = cvs/*@SU.SE
        broken_des3_mic = host/*@E.KTH.SE
        correct_des3_mic = host/*@SU.SE
        require_mechlist_mic = host/*@SU.SE
```

BUGS

All of 0.5.x versions of **heimdal** had broken token delegations in the client side, the server side was correct.

SEE ALSO

krb5(3), krb5.conf(5), kerberos(8)

NAME

hash — hash database access method

SYNOPSIS

```
#include <sys/types.h>
#include <db.h>
```

DESCRIPTION

The routine **dbopen**() is the library interface to database files. One of the supported file formats is hash files. The general description of the database access methods is in dbopen(3), this manual page describes only the hash specific information.

The hash data structure is an extensible, dynamic hashing scheme.

The access method specific data structure provided to **dbopen**() is defined in the ⟨db.h⟩ include file as follows:

```
typedef struct {
        u_int bsize;
        u_int ffactor;
        u_int nelem;
        u_int cachesize;
        uint32_t (*hash)(const void *, size_t);
        int lorder;
} HASHINFO;
```

The elements of this structure are as follows:

bsize *bsize* defines the hash table bucket size, and is, by default, 256 bytes. It may be preferable to increase the page size for disk-resident tables and tables with large data items.

ffactor *ffactor* indicates a desired density within the hash table. It is an approximation of the number of keys allowed to accumulate in any one bucket, determining when the hash table grows or shrinks. The default value is 8.

nelem *nelem* is an estimate of the final size of the hash table. If not set or set too low, hash tables will expand gracefully as keys are entered, although a slight performance degradation may be noticed. The default value is 1.

cachesize A suggested maximum size, in bytes, of the memory cache. This value is *only* advisory, and the access method will allocate more memory rather than fail.

hash *hash* is a user defined hash function. Since no hash function performs equally well on all possible data, the user may find that the built-in hash function does poorly on a particular data set. User specified hash functions must take two arguments (a pointer to a byte string and a length) and return a 32-bit quantity to be used as the hash value.

lorder The byte order for integers in the stored database metadata. The number should represent the order as an integer; for example, big endian order would be the number 4,321. If *lorder* is 0 (no order is specified) the current host order is used. If the file already exists, the specified value is ignored and the value specified when the tree was created is used.

If the file already exists (and the O_TRUNC flag is not specified), the values specified for the parameters *bsize*, *ffactor*, *lorder*, and *nelem* are ignored and the values specified when the tree was created are used.

If a hash function is specified, **hash_open**() will attempt to determine if the hash function specified is the same as the one with which the database was created, and will fail if it is not.

ERRORS

The **hash** access method routines may fail and set *errno* for any of the errors specified for the library routine dbopen(3).

SEE ALSO

btree(3), dbopen(3), mpool(3), recno(3)

Per-Ake Larson, "Dynamic Hash Tables", *Communications of the ACM*, April 1988.

Margo Seltzer, "A New Hash Package for UNIX", *USENIX Proceedings*, Winter 1991.

BUGS

Only big and little endian byte order is supported.

NAME

hcreate, **hdestroy**, **hsearch** — manage hash search table

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <search.h>

int
hcreate(size_t nel);

void
hdestroy(void);

ENTRY *
hsearch(ENTRY item, ACTION action);
```

DESCRIPTION

The **hcreate**(), **hdestroy**() and **hsearch**() functions manage hash search tables.

The **hcreate**() function allocates and initializes the table. The *nel* argument specifies an estimate of the maximum number of entries to be held by the table. Unless further memory allocation fails, supplying an insufficient *nel* value will not result in functional harm, although a performance degradation may occur. Initialization using the **hcreate**() function is mandatory prior to any access operations using **hsearch**().

The **hdestroy**() function destroys a table previously created using **hcreate**(). After a call to **hdestroy**(), the data can no longer be accessed.

The **hsearch**() function is used to search to the hash table. It returns a pointer into the hash table indicating the address of an item. The *item* argument is of type ENTRY, a structural type which contains the following members:

*char *key*	comparison key.
*void *data*	pointer to data associated with *key*.

The key comparison function used by **hsearch**() is strcmp(3).

The *action* argument is of type ACTION, an enumeration type which defines the following values:

ENTER	Insert *item* into the hash table. If an existing item with the same key is found, it is not replaced. Note that the *key* and *data* elements of *item* are used directly by the new table entry. The storage for the key must not be modified during the lifetime of the hash table.
FIND	Search the hash table without inserting *item*.

RETURN VALUES

If successful, the **hcreate**() function returns a non-zero value. Otherwise, a value of 0 is returned and *errno* is set to indicate the error.

The **hdestroy**() functions returns no value.

If successful, the **hsearch**() function returns a pointer to hash table entry matching the provided key. If the action is FIND and the item was not found, or if the action is ENTER and the insertion failed, NULL is returned and *errno* is set to indicate the error. If the action is ENTER and an entry already existed in the table matching the given key, the existing entry is returned and is not replaced.

ERRORS

The **hcreate**() and **hsearch**() functions will fail if:

[ENOMEM] Insufficient memory is available.

SEE ALSO

bsearch(3), lsearch(3), malloc(3), strcmp(3)

STANDARDS

The **hcreate**(), **hdestroy**() and **hsearch**() functions conform to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

HISTORY

The **hcreate**(), **hdestroy**() and **hsearch**() functions first appeared in AT&T System V UNIX.

BUGS

The interface permits the use of only one hash table at a time.

NAME

hesiod, **hesiod_init**, **hesiod_resolve**, **hesiod_free_list**, **hesiod_to_bind**, **hesiod_end** — Hesiod name server interface library

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <hesiod.h>

int
hesiod_init(void **context);

char
**hesiod_resolve(void *context, const char *name, const char *type);

void
hesiod_free_list(void *context, char **list);

char
*hesiod_to_bind(void *context, const char *name, const char *type);

void
hesiod_end(void *context);
```

DESCRIPTION

This family of functions allows you to perform lookups of Hesiod information, which is stored as text records in the Domain Name Service. To perform lookups, you must first initialize a *context*, an opaque object which stores information used internally by the library between calls. **hesiod_init**() initializes a context, storing a pointer to the context in the location pointed to by the *context* argument. **hesiod_end**() frees the resources used by a context.

hesiod_resolve() is the primary interface to the library. If successful, it returns a list of one or more strings giving the records matching *name* and *type*. The last element of the list is followed by a NULL pointer. It is the caller's responsibility to call **hesiod_free_list**() to free the resources used by the returned list.

hesiod_to_bind() converts *name* and *type* into the DNS name used by **hesiod_resolve**(). It is the caller's responsibility to free the returned string using free(3).

RETURN VALUES

If successful, **hesiod_init**() returns 0; otherwise it returns −1 and sets *errno* to indicate the error. On failure, **hesiod_resolve**() and **hesiod_to_bind**() return NULL and set the global variable *errno* to indicate the error.

ENVIRONMENT

If the environment variable HES_DOMAIN is set, it will override the domain in the Hesiod configuration file. If the environment variable HESIOD_CONFIG is set, it specifies the location of the Hesiod configuration file.

ERRORS

Hesiod calls may fail because of:

ENOMEM Insufficient memory was available to carry out the requested operation.

ENOEXEC **hesiod_init**() failed because the Hesiod configuration file was invalid.

ECONNREFUSED **hesiod_resolve**() failed because no name server could be contacted to answer the query.

EMSGSIZE **hesiod_resolve**() or **hesiod_to_bind**() failed because the query or response was too big to fit into the packet buffers.

ENOENT **hesiod_resolve**() failed because the name server had no text records matching *name* and *type*, or **hesiod_to_bind**() failed because the *name* argument had a domain extension which could not be resolved with type "rhs-extension" in the local Hesiod domain.

SEE ALSO

hesiod.conf(5), named(8)

Hesiod - Project Athena Technical Plan -- Name Service.

AUTHORS

Steve Dyer, IBM/Project Athena
Greg Hudson, MIT Team Athena

BUGS

The strings corresponding to the errno values set by the Hesiod functions are not particularly indicative of what went wrong, especially for ENOEXEC and ENOENT.

NAME

history − GNU History Library

COPYRIGHT

DESCRIPTION

Many programs read input from the user a line at a time. The GNU History library is able to keep track of those lines, associate arbitrary data with each line, and utilize information from previous lines in composing new ones.

HISTORY EXPANSION

The history library supports a history expansion feature that is identical to the history expansion in **bash.** This section describes what syntax features are available.

History expansions introduce words from the history list into the input stream, making it easy to repeat commands, insert the arguments to a previous command into the current input line, or fix errors in previous commands quickly.

History expansion is usually performed immediately after a complete line is read. It takes place in two parts. The first is to determine which line from the history list to use during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the history is the *event*, and the portions of that line that are acted upon are *words*. Various *modifiers* are available to manipulate the selected words. The line is broken into words in the same fashion as **bash** does when reading input, so that several words that would otherwise be separated are considered one word when surrounded by quotes (see the description of **history_tokenize()** below). History expansions are introduced by the appearance of the history expansion character, which is **!** by default. Only backslash (\) and single quotes can quote the history expansion character.

Event Designators

An event designator is a reference to a command line entry in the history list.

! Start a history substitution, except when followed by a **blank**, newline, = or (.

!*n* Refer to command line *n*.

!−*n* Refer to the current command line minus *n*.

!! Refer to the previous command. This is a synonym for '!−1'.

!*string* Refer to the most recent command starting with *string*.

!?*string***[?]**
Refer to the most recent command containing *string*. The trailing **?** may be omitted if *string* is followed immediately by a newline.

^*string1***^***string2***^**
Quick substitution. Repeat the last command, replacing *string1* with *string2*. Equivalent to ''!!:s/*string1*/*string2*/'' (see **Modifiers** below).

!# The entire command line typed so far.

Word Designators

Word designators are used to select desired words from the event. A **:** separates the event specification from the word designator. It may be omitted if the word designator begins with a **^**, **$**, *****, **−**, or **%**. Words are numbered from the beginning of the line, with the first word being denoted by 0 (zero). Words are inserted into the current line separated by single spaces.

0 (zero)
The zeroth word. For the shell, this is the command word.

n The *n*th word.

^ The first argument. That is, word 1.

$ The last argument.

% The word matched by the most recent '?*string*?' search.

x−*y* A range of words; '−*y*' abbreviates '0−*y*'.

***** All of the words but the zeroth. This is a synonym for '*1−$*'. It is not an error to use ***** if there is just one word in the event; the empty string is returned in that case.

x* Abbreviates *x−$*.

x− Abbreviates *x−$* like **x***, but omits the last word.

If a word designator is supplied without an event specification, the previous command is used as the event.

Modifiers

After the optional word designator, there may appear a sequence of one or more of the following modifiers, each preceded by a ':'.

h Remove a trailing file name component, leaving only the head.

t Remove all leading file name components, leaving the tail.

r Remove a trailing suffix of the form *.xxx*, leaving the basename.

e Remove all but the trailing suffix.

p Print the new command but do not execute it.

q Quote the substituted words, escaping further substitutions.

x Quote the substituted words as with **q**, but break into words at **blanks** and newlines.

s/*old*/*new*/

Substitute *new* for the first occurrence of *old* in the event line. Any delimiter can be used in place of /. The final delimiter is optional if it is the last character of the event line. The delimiter may be quoted in *old* and *new* with a single backslash. If & appears in *new*, it is replaced by *old*. A single backslash will quote the &. If *old* is null, it is set to the last *old* substituted, or, if no previous history substitutions took place, the last *string* in a **!?***string***[?]** search.

& Repeat the previous substitution.

g Cause changes to be applied over the entire event line. This is used in conjunction with '**:s**' (e.g., '**:gs**/*old*/*new*/') or '**:&**'. If used with '**:s**', any delimiter can be used in place of /, and the final delimiter is optional if it is the last character of the event line. An **a** may be used as a synonym for **g**.

G Apply the following '**s**' modifier once to each word in the event line.

PROGRAMMING WITH HISTORY FUNCTIONS

This section describes how to use the History library in other programs.

Introduction to History

The programmer using the History library has available functions for remembering lines on a history list, associating arbitrary data with a line, removing lines from the list, searching through the list for a line containing an arbitrary text string, and referencing any line in the list directly. In addition, a history *expansion* function is available which provides for a consistent user interface across different programs.

The user using programs written with the History library has the benefit of a consistent user interface with a set of well-known commands for manipulating the text of previous lines and using that text in new commands. The basic history manipulation commands are identical to the history substitution provided by **bash**.

If the programmer desires, he can use the Readline library, which includes some history manipulation by default, and has the added advantage of command line editing.

Before declaring any functions using any functionality the History library provides in other code, an application writer should include the file *<readline/history.h>* in any file that uses the History library's features. It supplies extern declarations for all of the library's public functions and variables, and declares all of the public data structures.

History Storage

The history list is an array of history entries. A history entry is declared as follows:

typedef void * **histdata_t;**

```
typedef struct _hist_entry {
  char *line;
```

```
  char *timestamp;
  histdata_t data;
} HIST_ENTRY;
```

The history list itself might therefore be declared as

HIST_ENTRY ** **the_history_list;**

The state of the History library is encapsulated into a single structure:

```
/*
 * A structure used to pass around the current state of the history.
 */
typedef struct _hist_state {
  HIST_ENTRY **entries; /* Pointer to the entries themselves. */
  int offset;          /* The location pointer within this array. */
  int length;          /* Number of elements within this array. */
  int size;            /* Number of slots allocated to this array. */
  int flags;
} HISTORY_STATE;
```

If the flags member includes **HS_STIFLED**, the history has been stifled.

History Functions

This section describes the calling sequence for the various functions exported by the GNU History library.

Initializing History and State Management

This section describes functions used to initialize and manage the state of the History library when you want to use the history functions in your program.

void **using_history** (*void*)
Begin a session in which the history functions might be used. This initializes the interactive variables.

HISTORY_STATE * **history_get_history_state** (*void*)
Return a structure describing the current state of the input history.

void **history_set_history_state** (*HISTORY_STATE *state*)
Set the state of the history list according to *state*.

History List Management

These functions manage individual entries on the history list, or set parameters managing the list itself.

void **add_history** (*const char *string*)
Place *string* at the end of the history list. The associated data field (if any) is set to **NULL**.

void **add_history_time** (*const char *string*)
Change the time stamp associated with the most recent history entry to *string*.

HIST_ENTRY * **remove_history** (*int which*)
Remove history entry at offset *which* from the history. The removed element is returned so you can free the line, data, and containing structure.

histdata_t **free_history_entry** (*HIST_ENTRY *histent*)
Free the history entry *histent* and any history library private data associated with it. Returns the application-specific data so the caller can dispose of it.

HIST_ENTRY * **replace_history_entry** (*int which, const char *line, histdata_t data*)
Make the history entry at offset *which* have *line* and *data*. This returns the old entry so the caller can

dispose of any application-specific data. In the case of an invalid *which*, a **NULL** pointer is returned.

void **clear_history** (*void*)
Clear the history list by deleting all the entries.

void **stifle_history** (*int max*)
Stifle the history list, remembering only the last *max* entries.

int **unstifle_history** (*void*)
Stop stifling the history. This returns the previously-set maximum number of history entries (as set by **stifle_history()**). history was stifled. The value is positive if the history was stifled, negative if it wasn't.

int **history_is_stifled** (*void*)
Returns non-zero if the history is stifled, zero if it is not.

Information About the History List

These functions return information about the entire history list or individual list entries.

HIST_ENTRY ** **history_list** (*void*)
Return a **NULL** terminated array of *HIST_ENTRY* * which is the current input history. Element 0 of this list is the beginning of time. If there is no history, return **NULL**.

int **where_history** (*void*)
Returns the offset of the current history element.

HIST_ENTRY * **current_history** (*void*)
Return the history entry at the current position, as determined by **where_history()**. If there is no entry there, return a **NULL** pointer.

HIST_ENTRY * **history_get** (*int offset*)
Return the history entry at position *offset*, starting from **history_base**. If there is no entry there, or if *offset* is greater than the history length, return a **NULL** pointer.

time_t **history_get_time** (*HIST_ENTRY* *)
Return the time stamp associated with the history entry passed as the argument.

int **history_total_bytes** (*void*)
Return the number of bytes that the primary history entries are using. This function returns the sum of the lengths of all the lines in the history.

Moving Around the History List

These functions allow the current index into the history list to be set or changed.

int **history_set_pos** (*int pos*)
Set the current history offset to *pos*, an absolute index into the list. Returns 1 on success, 0 if *pos* is less than zero or greater than the number of history entries.

HIST_ENTRY * **previous_history** (*void*)
Back up the current history offset to the previous history entry, and return a pointer to that entry. If there is no previous entry, return a **NULL** pointer.

HIST_ENTRY * **next_history** (*void*)
Move the current history offset forward to the next history entry, and return the a pointer to that entry. If there is no next entry, return a **NULL** pointer.

Searching the History List

These functions allow searching of the history list for entries containing a specific string. Searching may be performed both forward and backward from the current history position. The search may be *anchored*, meaning that the string must match at the beginning of the history entry.

int **history_search** (*const char *string, int direction*)
Search the history for *string*, starting at the current history offset. If *direction* is less than 0, then the search is through previous entries, otherwise through subsequent entries. If *string* is found, then the current history index is set to that history entry, and the value returned is the offset in the line of the entry where *string* was found. Otherwise, nothing is changed, and a -1 is returned.

int **history_search_prefix** (*const char *string, int direction*)
Search the history for *string*, starting at the current history offset. The search is anchored: matching lines must begin with *string*. If *direction* is less than 0, then the search is through previous entries, otherwise through subsequent entries. If *string* is found, then the current history index is set to that entry, and the return value is 0. Otherwise, nothing is changed, and a -1 is returned.

int **history_search_pos** (*const char *string, int direction, int pos*)
Search for *string* in the history list, starting at *pos*, an absolute index into the list. If *direction* is negative, the search proceeds backward from *pos*, otherwise forward. Returns the absolute index of the history element where *string* was found, or -1 otherwise.

Managing the History File

The History library can read the history from and write it to a file. This section documents the functions for managing a history file.

int **read_history** (*const char *filename*)
Add the contents of *filename* to the history list, a line at a time. If *filename* is **NULL**, then read from *~/.history*. Returns 0 if successful, or **errno** if not.

int **read_history_range** (*const char *filename, int from, int to*)
Read a range of lines from *filename*, adding them to the history list. Start reading at line *from* and end at *to*. If *from* is zero, start at the beginning. If *to* is less than *from*, then read until the end of the file. If *filename* is **NULL**, then read from *~/.history*. Returns 0 if successful, or **errno** if not.

int **write_history** (*const char *filename*)
Write the current history to *filename*, overwriting *filename* if necessary. If *filename* is **NULL**, then write the history list to *~/.history*. Returns 0 on success, or **errno** on a read or write error.

int **append_history** (*int nelements, const char *filename*)
Append the last *nelements* of the history list to *filename*. If *filename* is **NULL**, then append to *~/.history*. Returns 0 on success, or **errno** on a read or write error.

int **history_truncate_file** (*const char *filename, int nlines*)
Truncate the history file *filename*, leaving only the last *nlines* lines. If *filename* is **NULL**, then *~/.history* is truncated. Returns 0 on success, or **errno** on failure.

History Expansion

These functions implement history expansion.

int **history_expand** (*char *string, char **output*)
Expand *string*, placing the result into *output*, a pointer to a string. Returns:

0 If no expansions took place (or, if the only change in the text was the removal of escape characters preceding the history expansion character);
1 if expansions did take place;
-1 if there was an error in expansion;
2 if the returned line should be displayed, but not executed, as with the **:p** modifier.

If an error ocurred in expansion, then *output* contains a descriptive error message.

char * **get_history_event** (*const char *string, int *cindex, int qchar*)
Returns the text of the history event beginning at *string* + **cindex*. **cindex* is modified to point to after the event specifier. At function entry, *cindex* points to the index into *string* where the history event specification begins. *qchar* is a character that is allowed to end the event specification in addition to the ''normal'' terminating characters.

char ** **history_tokenize** (*const char *string*)
Return an array of tokens parsed out of *string*, much as the shell might. The tokens are split on the characters in the **history_word_delimiters** variable, and shell quoting conventions are obeyed.

char * **history_arg_extract** (*int first, int last, const char *string*)
Extract a string segment consisting of the *first* through *last* arguments present in *string*. Arguments are split using **history_tokenize()**.

History Variables

This section describes the externally-visible variables exported by the GNU History Library.

int **history_base**
The logical offset of the first entry in the history list.

int **history_length**
The number of entries currently stored in the history list.

int **history_max_entries**
The maximum number of history entries. This must be changed using **stifle_history()**.

int **history_write_timestamps**
If non-zero, timestamps are written to the history file, so they can be preserved between sessions. The default value is 0, meaning that timestamps are not saved.

char **history_expansion_char**
The character that introduces a history event. The default is **!**. Setting this to 0 inhibits history expansion.

char **history_subst_char**
The character that invokes word substitution if found at the start of a line. The default is **^**.

char **history_comment_char**
During tokenization, if this character is seen as the first character of a word, then it and all subsequent characters up to a newline are ignored, suppressing history expansion for the remainder of the line. This is disabled by default.

char * **history_word_delimiters**
The characters that separate tokens for **history_tokenize()**. The default value is **" \t\n()<>;&|"**.

char * **history_no_expand_chars**
The list of characters which inhibit history expansion if found immediately following **history_expansion_char**. The default is space, tab, newline, **\r**, and **=**.

char * **history_search_delimiter_chars**
The list of additional characters which can delimit a history search string, in addition to space, tab, *:* and *?* in the case of a substring search. The default is empty.

int **history_quotes_inhibit_expansion**
If non-zero, single-quoted words are not scanned for the history expansion character. The default value is 0.

rl_linebuf_func_t * **history_inhibit_expansion_function**
This should be set to the address of a function that takes two arguments: a **char *** (*string*) and an **int** index into that string (*i*). It should return a non-zero value if the history expansion starting at *string[i]* should not be performed; zero if the expansion should be done. It is intended for use by applications like **bash** that use the history expansion character for additional purposes. By default, this variable is set to **NULL**.

FILES

~/.history
Default filename for reading and writing saved history

SEE ALSO

The Gnu Readline Library, Brian Fox and Chet Ramey
The Gnu History Library, Brian Fox and Chet Ramey
bash(1)
readline(3)

AUTHORS

Brian Fox, Free Software Foundation
bfox@gnu.org

Chet Ramey, Case Western Reserve University
chet@ins.CWRU.Edu

BUG REPORTS

If you find a bug in the **history** library, you should report it. But first, you should make sure that it really is a bug, and that it appears in the latest version of the **history** library that you have.

Once you have determined that a bug actually exists, mail a bug report to *bug−readline@gnu.org*. If you have a fix, you are welcome to mail that as well! Suggestions and ‘philosophical’ bug reports may be mailed to *bug-readline@gnu.org* or posted to the Usenet newsgroup **gnu.bash.bug**.

Comments and bug reports concerning this manual page should be directed to *chet@ins.CWRU.Edu*.

NAME

hosts_access, hosts_ctl, request_init, request_set − access control library

SYNOPSIS

```
#include "tcpd.h"

extern int allow_severity;
extern int deny_severity;

struct request_info *request_init(request, key, value, ..., 0)
struct request_info *request;

struct request_info *request_set(request, key, value, ..., 0)
struct request_info *request;

int hosts_access(request)
struct request_info *request;

int hosts_ctl(daemon, client_name, client_addr, client_user)
char *daemon;
char *client_name;
char *client_addr;
char *client_user;
```

DESCRIPTION

The routines described in this document are part of the *libwrap.a* library. They implement a rule-based access control language with optional shell commands that are executed when a rule fires.

request_init() initializes a structure with information about a client request. request_set() updates an already initialized request structure. Both functions take a variable-length list of key-value pairs and return their first argument. The argument lists are terminated with a zero key value. All string-valued arguments are copied. The expected keys (and corresponding value types) are:

RQ_FILE (int)
: The file descriptor associated with the request.

RQ_CLIENT_NAME (char *)
: The client host name.

RQ_CLIENT_ADDR (char *)
: A printable representation of the client network address.

RQ_CLIENT_SIN (struct sockaddr_in *)
: An internal representation of the client network address and port. The contents of the structure are not copied.

RQ_SERVER_NAME (char *)
: The hostname associated with the server endpoint address.

RQ_SERVER_ADDR (char *)
: A printable representation of the server endpoint address.

RQ_SERVER_SIN (struct sockaddr_in *)
: An internal representation of the server endpoint address and port. The contents of the structure are not copied.

RQ_DAEMON (char *)
: The name of the daemon process running on the server host.

RQ_USER (char *)
: The name of the user on whose behalf the client host makes the request.

hosts_access() consults the access control tables described in the *hosts_access(5)* manual page. When internal endpoint information is available, host names and client user names are looked up on demand, using the request structure as a cache. hosts_access() returns zero if access should be denied.

hosts_ctl() is a wrapper around the request_init() and hosts_access() routines with a perhaps more convenient interface (though it does not pass on enough information to support automated client username lookups). The client host address, client host name and username arguments should contain valid data or STRING_UNKNOWN. hosts_ctl() returns zero if access should be denied.

The *allow_severity* and *deny_severity* variables determine how accepted and rejected requests may be logged. They must be provided by the caller and may be modified by rules in the access control tables.

DIAGNOSTICS

Problems are reported via the syslog daemon.

SEE ALSO

hosts_access(5), format of the access control tables. hosts_options(5), optional extensions to the base language.

FILES

/etc/hosts.allow, /etc/hosts.deny, access control tables.

AUTHOR

Wietse Venema (wietse@wzv.win.tue.nl)
Department of Mathematics and Computing Science
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513,
5600 MB Eindhoven, The Netherlands

NAME

dehumanize_number, **humanize_number** — format a number into a human readable form and vice-versa

SYNOPSIS

```
#include <stdlib.h>

int
dehumanize_number(const char *str, int64_t *result);

int
humanize_number(char *buf, size_t len, int64_t number, const char *suffix,
    int scale, int flags);
```

DESCRIPTION

The **humanize_number**() function formats the signed 64 bit quantity given in *number* into *buffer*. A space and then *suffix* is appended to the end. *buffer* must be at least *len* bytes long.

If the formatted number (including *suffix*) would be too long to fit into *buffer*, then divide *number* by 1024 until it will. In this case, prefix *suffix* with the appropriate SI designator.

The prefixes are:

Prefix	Description	Multiplier
k	kilo	1024
M	mega	1048576
G	giga	1073741824
T	tera	1099511627776
P	peta	1125899906842624
E	exa	1152921504606846976

len must be at least 4 plus the length of *suffix*, in order to ensure a useful result is generated into *buffer*. To use a specific prefix, specify this as *scale* (Multiplier = 1024 ^ scale). This can not be combined with any of the *scale* flags below.

The following flags may be passed in `scale`:

`HN_AUTOSCALE`
: Format the buffer using the lowest multiplier possible.

`HN_GETSCALE`
: Return the prefix index number (the number of times *number* must be divided to fit) instead of formatting it to the buffer.

The following flags may be passed in `flags`:

`HN_DECIMAL` If the final result is less than 10, display it using one digit.

`HN_NOSPACE` Do not put a space between *number* and the prefix.

`HN_B` Use 'B' (bytes) as prefix if the original result does not have a prefix.

`HN_DIVISOR_1000`
: Divide *number* with 1000 instead of 1024.

The **dehumanize_number**() function parses the string representing an integral value given in *str* and stores the numerical value in the integer pointed to by *result*. The provided string may hold one of the suffixes, which will be interpreted and used to scale up its accompanying numerical value.

RETURN VALUES

humanize_number() returns the number of characters stored in *buffer* (excluding the terminating NUL) upon success, or −1 upon failure. If HN_GETSCALE is specified, the prefix index number will be returned instead.

dehumanize_number() returns 0 if the string was parsed correctly. A −1 is returned to indicate failure and an error code is stored in *errno*.

ERRORS

dehumanize_number() will fail and no number will be stored in *result* if:

[EINVAL] The string in *str* was empty or carried an unknown suffix.

[ERANGE] The string in *str* represented a number that does not fit in *result*.

SEE ALSO

humanize_number(9)

HISTORY

humanize_number() first appeared in NetBSD 2.0.

dehumanize_number() first appeared in NetBSD 5.0.

NAME

hypot, **hypotf** — Euclidean distance and complex absolute value functions

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
hypot(double x, double y);

float
hypotf(float x, float y);
```

DESCRIPTION

The **hypot**() functions compute the sqrt(x*x+y*y) in such a way that underflow will not happen, and overflow occurs only if the final result deserves it.

hypot(∞, *v*) = **hypot**(*v*, ∞) = +∞ for all *v*, including *NaN*.

ERRORS

Below 0.97 *ulps*. Consequently **hypot**(*5.0*, *12.0*) = 13.0 exactly; in general, hypot returns an integer whenever an integer might be expected.

The same cannot be said for the shorter and faster version of hypot that is provided in the comments in cabs.c; its error can exceed 1.2 *ulps*.

NOTES

As might be expected, **hypot**(*v*, *NaN*) and **hypot**(*NaN*, *v*) are *NaN* for all *finite v*; with "reserved operand" in place of "*NaN*", the same is true on a VAX. But programmers on machines other than a VAX (it has no ∞) might be surprised at first to discover that **hypot**(±∞, *NaN*) = +∞. This is intentional; it happens because **hypot**(∞, *v*) = +∞ for *all v*, finite or infinite. Hence **hypot**(∞, *v*) is independent of *v*. Unlike the reserved operand fault on a VAX, the IEEE *NaN* is designed to disappear when it turns out to be irrelevant, as it does in **hypot**(∞, *NaN*).

SEE ALSO

math(3), sqrt(3)

HISTORY

Both a **hypot**() function and a **cabs**() function appeared in Version 7 AT&T UNIX. **cabs**() was removed from public namespace in NetBSD 5.0 to avoid conflicts with the complex function in C99.

NAME

iconv_open, **iconv_close**, **iconv** — codeset conversion functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <iconv.h>

iconv_t
iconv_open(const char *dstname, const char *srcname);

int
iconv_close(iconv_t cd);

size_t
iconv(iconv_t cd, const char ** restrict src, size_t * restrict srcleft,
    char ** restrict dst, size_t * restrict dstleft);
```

DESCRIPTION

The **iconv_open**() function opens a converter from the codeset *srcname* to the codeset *dstname* and returns its descriptor.

The **iconv_close**() function closes the specified converter *cd*.

The **iconv**() function converts the string in the buffer ∗*src* of length ∗*srcleft* bytes and stores the converted string in the buffer ∗*dst* of size ∗*dstleft* bytes. After calling **iconv**(), the values pointed to by *src*, *srcleft*, *dst*, and *dstleft* are updated as follows:

∗src Pointer to the byte just after the last character fetched.

∗srcleft Number of remaining bytes in the source buffer.

∗dst Pointer to the byte just after the last character stored.

∗dstleft Number of remainder bytes in the destination buffer.

If the string pointed to by ∗*src* contains a byte sequence which is not a valid character in the source codeset, the conversion stops just after the last successful conversion. If the output buffer is too small to store the converted character, the conversion also stops in the same way. In these cases, the values pointed to by *src*, *srcleft*, *dst*, and *dstleft* are updated to the state just after the last successful conversion.

If the string pointed to by ∗*src* contains a character which is valid under the source codeset but can not be converted to the destination codeset, the character is replaced by an “invalid character” which depends on the destination codeset, e.g., ‘?’, and the conversion is continued. **iconv**() returns the number of such “invalid conversions”.

There are two special cases of **iconv**():

src == NULL || ∗src == NULL
: If the source and/or destination codesets are stateful, **iconv**() places these into their initial state.

  If both *dst* and ∗*dst* are non-NULL, **iconv**() stores the shift sequence for the destination switching to the initial state in the buffer pointed to by ∗*dst*. The buffer size is specified by the value pointed to by *dstleft* as above. **iconv**() will fail if the buffer is too small to store the shift sequence.

  On the other hand, *dst* or ∗*dst* may be NULL. In this case, the shift sequence for the destination switching to the initial state is discarded.

RETURN VALUES

Upon successful completion of **iconv_open**(), it returns a conversion descriptor. Otherwise, **iconv_open**() returns (iconv_t)−1 and sets errno to indicate the error.

Upon successful completion of **iconv_close**(), it returns 0. Otherwise, **iconv_close**() returns −1 and sets errno to indicate the error.

Upon successful completion of **iconv**(), it returns the number of "invalid" conversions. Otherwise, **iconv**() returns (size_t)−1 and sets errno to indicate the error.

ERRORS

The **iconv_open**() function may cause an error in the following cases:

[ENOMEM] Memory is exhausted.

[EINVAL] There is no converter specified by *srcname* and *dstname*.

The **iconv_close**() function may cause an error in the following case:

[EBADF] The conversion descriptor specified by *cd* is invalid.

The **iconv**() function may cause an error in the following cases:

[EBADF] The conversion descriptor specified by *cd* is invalid.

[EILSEQ] The string pointed to by ∗*src* contains a byte sequence which does not describe a valid character of the source codeset.

[E2BIG] The output buffer pointed to by ∗*dst* is too small to store the result string.

[EINVAL] The string pointed to by ∗*src* terminates with an incomplete character or shift sequence.

SEE ALSO

iconv(1)

STANDARDS

iconv_open(), **iconv_close**(), and **iconv**() conform to IEEE Std 1003.1-2001 ("POSIX.1").

BUGS

If **iconv**() is aborted due to the occurrence of some error, the "invalid conversion" count mentioned above is unfortunately lost.

NAME

copysign, **copysignf**, **finite**, **finitef**, **ilogb**, **ilogbf**, **nextafter**, **nextafterf**, **remainder**, **remainderf**, **scalbn**, **scalbnf** — functions for IEEE arithmetic

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
copysign(double x, double y);

float
copysignf(float x, float y);

int
finite(double x);

int
finitef(float x);

int
ilogb(double x);

int
ilogbf(float x);

double
nextafter(double x, double y);

float
nextafterf(float x, float y);

double
remainder(double x, double y);

float
remainderf(float x, float y);

double
scalbn(double x, int n);

float
scalbnf(float x, int n);
```

DESCRIPTION

These functions are required or recommended by IEEE Std 754-1985.

copysign() returns *x* with its sign changed to *y*'s.

finite() returns the value 1 just when $-\infty < x < +\infty$; otherwise a zero is returned (when $|x| = \infty$ or *x* is *NaN*).

ilogb() returns *x*'s exponent *n*, in integer format. **ilogb**(±∞) returns INT_MAX and **ilogb**(*0*) returns INT_MIN.

nextafter() returns the next machine representable number from *x* in direction *y*.

remainder() returns the remainder $r := x - n*y$ where n is the integer nearest the exact value of x/y; moreover if $|n - x/y| = 1/2$ then n is even. Consequently the remainder is computed exactly and $|r| \leq |y|/2$. But **remainder**(x, 0) and **remainder**(∞, 0) are invalid operations that produce a *NaN*.

scalbn() returns $x*(2**n)$ computed by exponent manipulation.

SEE ALSO

math(3)

STANDARDS

IEEE Std 754-1985

HISTORY

The **ieee** functions appeared in 4.3BSD.

NAME

logb, **logbf**, **scalb**, **scalbf**, **significand**, **significandf** — IEEE test functions

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
logb(double x);

float
logbf(float x);

double
scalb(double x, double n);

float
scalbf(float x, float n);

double
significand(double x);

float
significandf(float x);
```

DESCRIPTION

These functions allow users to test conformance to IEEE Std 754-1985. Their use is not otherwise recommended.

logb(*x*) returns *x*'s exponent *n*, a signed integer converted to double−precision floating−point. **logb**(±∞) = +∞; **logb**(*0*) = -∞ with a division by zero exception.

scalbn(*x*, *n*) returns *x*∗(2∗∗*n*) computed by exponent manipulation.

significand(*x*) returns *sig*, where *x* := *sig* ∗ 2∗∗*n* with 1 ≤ *sig* < 2. **significand**(*x*) is not defined when *x* is 0, ±∞, or *NaN*.

SEE ALSO

ieee(3), math(3)

STANDARDS

IEEE Std 754-1985

NAME

if_nametoindex, **if_indextoname**, **if_nameindex**, **if_freenameindex** — provide mappings between interface names and indexes

SYNOPSIS

```
#include <net/if.h>

unsigned int
if_nametoindex(const char *ifname);

char *
if_indextoname(unsigned int ifindex, char *ifname);

struct if_nameindex *
if_nameindex(void);

void
if_freenameindex(struct if_nameindex *ptr);
```

DESCRIPTION

The **if_nametoindex**() function maps the interface name specified in *ifname* to its corresponding index. If the specified interface does not exist, it returns 0.

The **if_indextoname**() function maps the interface index specified in *ifindex* to it corresponding name, which is copied into the buffer pointed to by *ifname*, which must be of at least IFNAMSIZ bytes. This pointer is also the return value of the function. If there is no interface corresponding to the specified index, NULL is returned.

The **if_nameindex**() function returns an array of **if_nameindex** structures, one structure per interface, as defined in the include file ⟨net/if.h⟩. The **if_nameindex** structure contains at least the following entries:

```
unsigned int   if_index;  /* 1, 2, ... */
char          *if_name;   /* null terminated name: "le0", ... */
```

The end of the array of structures is indicated by a structure with an **if_index** of 0 and an **if_name** of NULL. A NULL pointer is returned upon an error.

The **if_freenameindex**() function frees the dynamic memory that was allocated by **if_nameindex**().

RETURN VALUES

Upon successful completion, **if_nametoindex**() returns the index number of the interface. If the interface is not found, a value of 0 is returned and *errno* is set to ENXIO. A value of 0 is also returned if an error occurs while retrieving the list of interfaces via getifaddrs(3).

Upon successful completion, **if_indextoname**() returns *ifname*. If the interface is not found, a NULL pointer is returned and *errno* is set to ENXIO. A NULL pointer is also returned if an error occurs while retrieving the list of interfaces via getifaddrs(3).

The **if_nameindex**() returns a NULL pointer if an error occurs while retrieving the list of interfaces via getifaddrs(3), or if sufficient memory cannot be allocated.

SEE ALSO

getifaddrs(3), networking(4)

STANDARDS

The **if_nametoindex**(), **if_indextoname**(), **if_nameindex**(), and **if_freenameindex**() functions conform to IEEE Std 1003.1-2001 (“POSIX.1”), X/Open Networking Services Issue 5.2 (“XNS5.2”), and RFC 3493.

HISTORY

The implementation first appeared in BSD/OS.

NAME

index — locate character in string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <strings.h>

char *
index(const char *s, int c);
```

DESCRIPTION

The **index**() function locates the first character matching *c* (converted to a *char*) in the nul-terminated string *s*.

RETURN VALUES

A pointer to the character is returned if it is found; otherwise NULL is returned. If *c* is '\0', **index**() locates the terminating '\0'.

SEE ALSO

memchr(3), rindex(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3)

HISTORY

An **index**() function appeared in Version 6 AT&T UNIX.

NAME

inet_addr, **inet_aton**, **inet_lnaof**, **inet_makeaddr**, **inet_netof**, **inet_network**, **inet_ntoa**, **inet_ntop**, **inet_pton**, **addr**, **ntoa**, **network** — Internet address manipulation routines

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <arpa/inet.h>

in_addr_t
inet_addr(const char *cp);

int
inet_aton(const char *cp, struct in_addr *addr);

in_addr_t
inet_lnaof(struct in_addr in);

struct in_addr
inet_makeaddr(in_addr_t net, in_addr_t lna);

in_addr_t
inet_netof(struct in_addr in);

in_addr_t
inet_network(const char *cp);

char *
inet_ntoa(struct in_addr in);

const char *
inet_ntop(int af, const void * restrict src, char * restrict dst,
    socklen_t size);

int
inet_pton(int af, const char * restrict src, void * restrict dst);
```

DESCRIPTION

The routines **inet_aton**(), **inet_addr**() and **inet_network**() interpret character strings representing numbers expressed in the Internet standard "dotted quad" notation.

The **inet_pton**() function converts a presentation format address (that is, printable form as held in a character string) to network format (usually a *struct in_addr* or some other internal binary representation, in network byte order). It returns 1 if the address was valid for the specified address family, or 0 if the address wasn't parsable in the specified address family, or -1 if some system error occurred (in which case *errno* will have been set). This function is presently valid for AF_INET and AF_INET6.

The **inet_aton**() routine interprets the specified character string as an Internet address, placing the address into the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid.

The **inet_addr**() and **inet_network**() functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively.

The function **inet_ntop**() converts an address from network format (usually a *struct in_addr* or some other binary form, in network byte order) to presentation format (suitable for external display purposes). It returns NULL if a system error occurs (in which case, *errno* will have been set), or it returns a

pointer to the destination string.

The routine **inet_ntoa**() takes an Internet address and returns an ASCII string representing the address in "dotted quad" notation.

The routine **inet_makeaddr**() takes an Internet network number and a local network address (both in host order) and constructs an Internet address from it. Note that to convert only a single value to a *struct in_addr* form that value should be passed as the first parameter and '0L' should be given for the second parameter.

The routines **inet_netof**() and **inet_lnaof**() break apart Internet host addresses, returning the network number and local network address part, respectively (both in host order).

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES (IP VERSION 4)

Values specified using the "dotted quad" notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on a system that uses little-endian byte order (e.g. Intel i386, i486 and Pentium processors) the bytes referred to above appear as "d.c.b.a". That is, little-endian bytes are ordered from right to left.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a "dotted quad" notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

INTERNET ADDRESSES (IP VERSION 6)

In order to support scoped IPv6 addresses, the use of getaddrinfo(3) and getnameinfo(3) is recommended rather than the functions presented here.

The presentation format of an IPv6 address is given in RFC 2373:

There are three conventional forms for representing IPv6 addresses as text strings:

1. The preferred form is x:x:x:x:x:x:x:x, where the 'x's are the hexadecimal values of the eight 16-bit pieces of the address. Examples:

```
FEDC:BA98:7654:3210:FEDC:BA98:7654:3210
1080:0:0:0:8:800:200C:417A
```

Note that it is not necessary to write the leading zeros in an individual field, but there must be at least one numeral in every field (except for the case described in 2).

2. Due to the method of allocating certain styles of IPv6 addresses, it will be common for addresses to contain long strings of zero bits. In order to make writing addresses containing zero bits easier, a special syntax is available to compress the zeros. The use of ‘‘::’’ indicates multiple groups of 16-bits of zeros. The ‘‘::’’ can only appear once in an address. The ‘‘::’’ can also be used to compress the leading and/or trailing zeros in an address.

   For example the following addresses:

```
1080:0:0:0:8:800:200C:417A  a unicast address
FF01:0:0:0:0:0:0:43         a multicast address
0:0:0:0:0:0:0:1             the loopback address
0:0:0:0:0:0:0:0             the unspecified addresses
```

   may be represented as:

```
1080::8:800:200C:417A       a unicast address
FF01::43                    a multicast address
::1                         the loopback address
::                          the unspecified addresses
```

3. An alternative form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is x:x:x:x:x:x:d.d.d.d, where the ’x’s are the hexadecimal values of the six high-order 16-bit pieces of the address, and the ’d’s are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation). Examples:

```
0:0:0:0:0:0:13.1.68.3
0:0:0:0:0:FFFF:129.144.52.38
```

   or in compressed form:

```
::13.1.68.3
::FFFF:129.144.52.38
```

DIAGNOSTICS

The constant `INADDR_NONE` is returned by **`inet_addr`**() and **`inet_network`**() for malformed requests.

SEE ALSO

`byteorder`(3), `gethostbyname`(3), `getnetent`(3), `inet_net`(3), `hosts`(5), `networks`(5)

IP Version 6 Addressing Architecture, RFC 2373, July 1998.

Basic Socket Interface Extensions for IPv6, RFC 3493, February 2003.

STANDARDS

The **`inet_ntop`** and **`inet_pton`** functions conform to IEEE Std 1003.1-2001 (“POSIX.1”). Note that **`inet_pton`** does not accept 1-, 2-, or 3-part dotted addresses; all four parts must be specified. This is a narrower input set than that accepted by **`inet_aton`**.

HISTORY

The **`inet_addr`**, **`inet_network`**, **`inet_makeaddr`**, **`inet_lnaof`** and **`inet_netof`** functions appeared in 4.2BSD. They were changed to use *in_addr_t* in place of *unsigned long* in NetBSD 2.0. The **`inet_aton`** and **`inet_ntoa`** functions appeared in 4.3BSD. The **`inet_pton`** and **`inet_ntop`** functions appeared in BIND 4.9.4 and thence NetBSD 1.3; they were also in X/Open Networking Services Issue 5.2 (“XNS5.2”).

BUGS

The value `INADDR_NONE` (0xffffffff) is a valid broadcast address, but **`inet_addr`**() cannot return that value without indicating failure. The newer **`inet_aton`**() function does not share this problem.

The problem of host byte ordering versus network byte ordering is confusing.

The string returned by **`inet_ntoa`**() resides in a static memory area.

`inet_addr`() should return a *`struct in_addr`*.

NAME

inet6_opt_init, **inet6_opt_append**, **inet6_opt_finish**, **inet6_opt_set_val**, **inet6_opt_next**, **inet6_opt_find**, **inet6_opt_get_val** — IPv6 Hop-by-Hop and Destination Options manipulation

SYNOPSIS

```
#include <netinet/in.h>

int
inet6_opt_init(void *extbuf, socklen_t extlen);

int
inet6_opt_append(void *extbuf, socklen_t extlen, int offset, u_int8_t type,
    socklen_t len, u_int8_t align, void **databufp);

int
inet6_opt_finish(void *extbuf, socklen_t extlen, int offset);

int
inet6_opt_set_val(void *databuf, int offset, void *val, socklen_t vallen);

int
inet6_opt_next(void *extbuf, socklen_t extlen, int offset, u_int8_t *typep,
    socklen_t *lenp, void **databufp);

int
inet6_opt_find(void *extbuf, socklen_t extlen, int offset, u_int8_t type,
    socklen_t *lenp, void **databufp);

int
inet6_opt_get_val(void *databuf, socklen_t offset, void *val,
    socklen_t vallen);
```

DESCRIPTION

Building and parsing the Hop-by-Hop and Destination options is complicated. The advanced sockets API defines a set of functions to help applications create and manipulate Hop-by-Hope and Destination options. These functions use the formatting rules specified in Appendix B in RFC2460, i.e., that the largest field is placed last in the option. The function prototypes for these functions are all contained in the <netinet/in.h> header file.

inet6_opt_init

The **inet6_opt_init**() function returns the number of bytes needed for an empty extension header, one without any options. If the *extbuf* argument points to a valid section of memory then the **inet6_opt_init**() function also initializes the extension header's length field. When attempting to initialize an extension buffer passed in the *extbuf argument* *extlen* must be a positive multiple of 8 or else the function fails and returns −1 to the caller.

inet6_opt_append

The **inet6_opt_append**() function can perform to different jobs. When a valid *extbuf* argument is supplied it appends an option to the extension buffer and returns the updated total length as well as a pointer to the newly created option in *databufp*. If the value of *extbuf* is NULL then the **inet6_opt_append**(*function*, *only*, *reports*, *what*, *the*, *total*, *length*, *would*) be if the option were actually appended. The *len* and *align* arguments specify the length of the option and the required data alignment which must be used when appending the option. The *offset* argument should be the length returned by the **inet6_opt_init**() function or a previous call to **inet6_opt_append**().

The *type* argument is the 8-bit option type.

After **inet6_opt_append**() has been called, the application can use the buffer pointed to by *databufp* directly, or use **inet6_opt_set_val**() to specify the data to be contained in the option.

Option types of `0` and `1` are reserved for the `Pad1` and `PadN` options. All other values from 2 through 255 may be used by applications.

The length of the option data is contained in an 8-bit value and so may contain any value from 0 through 255.

The *align* parameter must have a value of 1, 2, 4, or 8 and cannot exceed the value of *len*. The alignment values represent no alignment, 16 bit, 32 bit and 64 bit alignments respectively.

inet6_opt_finish

The **inet6_opt_finish**() calculates the final padding necessary to make the extension header a multiple of 8 bytes, as required by the IPv6 extension header specification, and returns the extension header's updated total length. The *offset* argument should be the length returned by **inet6_opt_init**() or **inet6_opt_append**(). When *extbuf* is not `NULL` the function also sets up the appropriate padding bytes by inserting a Pad1 or PadN option of the proper length.

If the extension header is too small to contain the proper padding then an error of −1 is returned to the caller.

inet6_opt_set_val

The **inet6_opt_set_val**() function inserts data items of various sizes into the data portion of the option. The *databuf* argument is a pointer to memory that was returned by the **inet6_opt_append**() call and the *offset argument specifies where the option should be placed in the* data buffer. The *val* argument points to an area of memory containing the data to be inserted into the extension header, and the *vallen* argument indicates how much data to copy.

The caller should ensure that each field is aligned on its natural boundaries as described in Appendix B of RFC2460.

The function returns the offset for the next field which is calculated as *offset* + *vallen* and is used when composing options with multiple fields.

inet6_opt_next

The **inet6_opt_next**() function parses received extension headers. The *extbuf* and *extlen* arguments specify the location and length of the extension header being parsed. The *offset* argument should either be zero, for the first option, or the length value returned by a previous call to **inet6_opt_next**() or **inet6_opt_find**(). The return value specifies the position where to continue scanning the extension buffer. The option is returned in the arguments *typep*, *lenp*, and *databufp*. *typep, lenp,* and *databufp* point to the 8-bit option type, the 8-bit option length and the option data respectively. This function does not return any PAD1 or PADN options. When an error occurs or there are no more options the return value is −1.

inet6_opt_find

The **inet6_opt_find**() function searches the extension buffer for a particular option type, passed in through the *type* argument. If the option is found then the *lenp* and *databufp* arguments are updated to point to the option's length and data respectively. *extbuf* and *extlen* must point to a valid extension buffer and give its length. The *offset* argument can be used to search from a location anywhere in the extension header.

inet6_opt_get_val

The **inet6_opt_get_val**() function extracts data items of various sizes in the data portion of the option. The *databuf* is a pointer returned by the **inet6_opt_next**() or **inet6_opt_find**() functions. The

val argument points where the data will be extracted. The *offset* argument specifies from where in the data portion of the option the value should be extracted; the first byte of option data is specified by an offset of zero.

It is expected that each field is aligned on its natural boundaries as described in Appendix B of RFC2460.

The function returns the offset for the next field by calculating *offset* + *vallen* which can be used when extracting option content with multiple fields. Robust receivers must verify alignment before calling this function.

DIAGNOSTICS

All the functions return −1 on an error.

EXAMPLES

RFC3542 gives comprehensive examples in Section 23.

KAME also provides examples in the `advapitest` directory of its kit.

SEE ALSO

W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei, *Advanced Sockets API for IPv6*, RFC3542, October 2002.

S. Deering and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC2460, December 1998.

HISTORY

The implementation first appeared in KAME advanced networking kit.

STANDARDS

The functions are documented in “Advanced Sockets API for IPv6” (RFC3542).

NAME

inet6_option_space, **inet6_option_init**, **inet6_option_append**, **inet6_option_alloc**, **inet6_option_next**, **inet6_option_find** — IPv6 Hop-by-Hop and Destination Options manipulation

SYNOPSIS

```
#include <netinet/in.h>

int
inet6_option_space(int nbytes);

int
inet6_option_init(void *bp, struct cmsghdr **cmsgp, int type);

int
inet6_option_append(struct cmsghdr *cmsg, const uint8_t *typep, int multx,
    int plusy);

uint8_t *
inet6_option_alloc(struct cmsghdr *cmsg, int datalen, int multx, int plusy);

int
inet6_option_next(const struct cmsghdr *cmsg, uint8_t **tptrp);

int
inet6_option_find(const struct cmsghdr *cmsg, uint8_t **tptrp, int type);
```

DESCRIPTION

Building and parsing the Hop-by-Hop and Destination options is complicated due to alignment constraints, padding and ancillary data manipulation. RFC 2292 defines a set of functions to help the application. The function prototypes for these functions are all in the ⟨`netinet/in.h`⟩ header.

inet6_option_space

inet6_option_space() returns the number of bytes required to hold an option when it is stored as ancillary data, including the `cmsghdr` structure at the beginning, and any padding at the end (to make its size a multiple of 8 bytes). The argument is the size of the structure defining the option, which must include any pad bytes at the beginning (the value `y` in the alignment term "`xn + y`"), the type byte, the length byte, and the option data.

Note: If multiple options are stored in a single ancillary data object, which is the recommended technique, this function overestimates the amount of space required by the size of `N-1` `cmsghdr` structures, where `N` is the number of options to be stored in the object. This is of little consequence, since it is assumed that most Hop-by-Hop option headers and Destination option headers carry only one option (appendix B of [RFC 2460]).

inet6_option_init

inet6_option_init() is called once per ancillary data object that will contain either Hop-by-Hop or Destination options. It returns `0` on success or `-1` on an error.

bp is a pointer to previously allocated space that will contain the ancillary data object. It must be large enough to contain all the individual options to be added by later calls to **inet6_option_append**() and **inet6_option_alloc**().

cmsgp is a pointer to a pointer to a `cmsghdr` structure. **cmsgp* is initialized by this function to point to the `cmsghdr` structure constructed by this function in the buffer pointed to by *bp*.

type is either IPV6_HOPOPTS or IPV6_DSTOPTS. This *type* is stored in the cmsg_type member of the cmsghdr structure pointed to by ∗*cmsgp*.

inet6_option_append

This function appends a Hop-by-Hop option or a Destination option into an ancillary data object that has been initialized by **inet6_option_init**(). This function returns 0 if it succeeds or -1 on an error.

cmsg is a pointer to the cmsghdr structure that must have been initialized by **inet6_option_init**().

typep is a pointer to the 8-bit option type. It is assumed that this field is immediately followed by the 8-bit option data length field, which is then followed immediately by the option data. The caller initializes these three fields (the type-length-value, or TLV) before calling this function.

The option type must have a value from 2 to 255, inclusive. (0 and 1 are reserved for the Pad1 and PadN options, respectively.)

The option data length must have a value between 0 and 255, inclusive, and is the length of the option data that follows.

multx is the value x in the alignment term "xn + y". It must have a value of 1, 2, 4, or 8.

plusy is the value y in the alignment term "xn + y". It must have a value between 0 and 7, inclusive.

inet6_option_alloc

This function appends a Hop-by-Hop option or a Destination option into an ancillary data object that has been initialized by **inet6_option_init**(). This function returns a pointer to the 8-bit option type field that starts the option on success, or NULL on an error.

The difference between this function and **inet6_option_append**() is that the latter copies the contents of a previously built option into the ancillary data object while the current function returns a pointer to the space in the data object where the option's TLV must then be built by the caller.

cmsg is a pointer to the cmsghdr structure that must have been initialized by **inet6_option_init**().

datalen is the value of the option data length byte for this option. This value is required as an argument to allow the function to determine if padding must be appended at the end of the option. (The **inet6_option_append**() function does not need a data length argument since the option data length must already be stored by the caller.)

multx is the value x in the alignment term "xn + y". It must have a value of 1, 2, 4, or 8.

plusy is the value y in the alignment term "xn + y". It must have a value between 0 and 7, inclusive.

inet6_option_next

This function processes the next Hop-by-Hop option or Destination option in an ancillary data object. If another option remains to be processed, the return value of the function is 0 and ∗*tptrp* points to the 8-bit option type field (which is followed by the 8-bit option data length, followed by the option data). If no more options remain to be processed, the return value is -1 and ∗*tptrp* is NULL. If an error occurs, the return value is -1 and ∗*tptrp* is not NULL.

cmsg is a pointer to cmsghdr structure of which cmsg_level equals IPPROTO_IPV6 and cmsg_type equals either IPV6_HOPOPTS or IPV6_DSTOPTS.

tptrp is a pointer to a pointer to an 8-bit byte and ∗*tptrp* is used by the function to remember its place in the ancillary data object each time the function is called. The first time this function is called for a given ancillary data object, ∗*tptrp* must be set to NULL.

Each time this function returns success, **tptrp* points to the 8-bit option type field for the next option to be processed.

inet6_option_find

This function is similar to the previously described **inet6_option_next**() function, except this function lets the caller specify the option type to be searched for, instead of always returning the next option in the ancillary data object. *cmsg* is a pointer to `cmsghdr` structure of which `cmsg_level` equals `IPPROTO_IPV6` and `cmsg_type` equals either `IPV6_HOPOPTS` or `IPV6_DSTOPTS`.

tptrp is a pointer to a pointer to an 8-bit byte and **tptrp* is used by the function to remember its place in the ancillary data object each time the function is called. The first time this function is called for a given ancillary data object, **tptrp* must be set to `NULL`. ~ This function starts searching for an option of the specified type beginning after the value of **tptrp*. If an option of the specified type is located, this function returns `0` and **tptrp* points to the 8- bit option type field for the option of the specified type. If an option of the specified type is not located, the return value is `-1` and **tptrp* is `NULL`. If an error occurs, the return value is `-1` and **tptrp* is not `NULL`.

EXAMPLES

RFC 2292 gives comprehensive examples in chapter 6.

DIAGNOSTICS

inet6_option_init() and **inet6_option_append**() return `0` on success or `-1` on an error.

inet6_option_alloc() returns `NULL` on an error.

On errors, **inet6_option_next**() and **inet6_option_find**() return `-1` setting **tptrp* to non `NULL` value.

SEE ALSO

W. Stevens and M. Thomas, *Advanced Sockets API for IPv6*, RFC 2292, February 1998.

S. Deering and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 2460, December 1998.

STANDARDS

The functions are documented in “Advanced Sockets API for IPv6” (RFC 2292) .

HISTORY

The implementation first appeared in KAME advanced networking kit.

BUGS

The text was shamelessly copied from RFC 2292.

NAME

inet6_option_space, **inet6_option_init**, **inet6_option_append**, **inet6_option_alloc**, **inet6_option_next**, **inet6_option_find** — IPv6 Hop-by-Hop and Destination Options manipulation

SYNOPSIS

```
#include <netinet/in.h>

int
inet6_option_space(int nbytes);

int
inet6_option_init(void *bp, struct cmsghdr **cmsgp, int type);

int
inet6_option_append(struct cmsghdr *cmsg, const uint8_t *typep, int multx,
    int plusy);

uint8_t *
inet6_option_alloc(struct cmsghdr *cmsg, int datalen, int multx, int plusy);

int
inet6_option_next(const struct cmsghdr *cmsg, uint8_t **tptrp);

int
inet6_option_find(const struct cmsghdr *cmsg, uint8_t **tptrp, int type);
```

DESCRIPTION

Building and parsing the Hop-by-Hop and Destination options is complicated due to alignment constraints, padding and ancillary data manipulation. RFC 2292 defines a set of functions to help the application. The function prototypes for these functions are all in the ⟨`netinet/in.h`⟩ header.

inet6_option_space

inet6_option_space() returns the number of bytes required to hold an option when it is stored as ancillary data, including the `cmsghdr` structure at the beginning, and any padding at the end (to make its size a multiple of 8 bytes). The argument is the size of the structure defining the option, which must include any pad bytes at the beginning (the value `y` in the alignment term "`xn + y`"), the type byte, the length byte, and the option data.

Note: If multiple options are stored in a single ancillary data object, which is the recommended technique, this function overestimates the amount of space required by the size of `N-1` `cmsghdr` structures, where `N` is the number of options to be stored in the object. This is of little consequence, since it is assumed that most Hop-by-Hop option headers and Destination option headers carry only one option (appendix B of [RFC 2460]).

inet6_option_init

inet6_option_init() is called once per ancillary data object that will contain either Hop-by-Hop or Destination options. It returns `0` on success or `-1` on an error.

bp is a pointer to previously allocated space that will contain the ancillary data object. It must be large enough to contain all the individual options to be added by later calls to **inet6_option_append**() and **inet6_option_alloc**().

cmsgp is a pointer to a pointer to a `cmsghdr` structure. **cmsgp* is initialized by this function to point to the `cmsghdr` structure constructed by this function in the buffer pointed to by *bp*.

type is either IPV6_HOPOPTS or IPV6_DSTOPTS. This *type* is stored in the cmsg_type member of the cmsghdr structure pointed to by ∗*cmsgp*.

inet6_option_append

This function appends a Hop-by-Hop option or a Destination option into an ancillary data object that has been initialized by **inet6_option_init**(). This function returns 0 if it succeeds or -1 on an error.

cmsg is a pointer to the cmsghdr structure that must have been initialized by **inet6_option_init**().

typep is a pointer to the 8-bit option type. It is assumed that this field is immediately followed by the 8-bit option data length field, which is then followed immediately by the option data. The caller initializes these three fields (the type-length-value, or TLV) before calling this function.

The option type must have a value from 2 to 255, inclusive. (0 and 1 are reserved for the Pad1 and PadN options, respectively.)

The option data length must have a value between 0 and 255, inclusive, and is the length of the option data that follows.

multx is the value x in the alignment term "xn + y". It must have a value of 1, 2, 4, or 8.

plusy is the value y in the alignment term "xn + y". It must have a value between 0 and 7, inclusive.

inet6_option_alloc

This function appends a Hop-by-Hop option or a Destination option into an ancillary data object that has been initialized by **inet6_option_init**(). This function returns a pointer to the 8-bit option type field that starts the option on success, or NULL on an error.

The difference between this function and **inet6_option_append**() is that the latter copies the contents of a previously built option into the ancillary data object while the current function returns a pointer to the space in the data object where the option's TLV must then be built by the caller.

cmsg is a pointer to the cmsghdr structure that must have been initialized by **inet6_option_init**().

datalen is the value of the option data length byte for this option. This value is required as an argument to allow the function to determine if padding must be appended at the end of the option. (The **inet6_option_append**() function does not need a data length argument since the option data length must already be stored by the caller.)

multx is the value x in the alignment term "xn + y". It must have a value of 1, 2, 4, or 8.

plusy is the value y in the alignment term "xn + y". It must have a value between 0 and 7, inclusive.

inet6_option_next

This function processes the next Hop-by-Hop option or Destination option in an ancillary data object. If another option remains to be processed, the return value of the function is 0 and ∗*tptrp* points to the 8-bit option type field (which is followed by the 8-bit option data length, followed by the option data). If no more options remain to be processed, the return value is -1 and ∗*tptrp* is NULL. If an error occurs, the return value is -1 and ∗*tptrp* is not NULL.

cmsg is a pointer to cmsghdr structure of which cmsg_level equals IPPROTO_IPV6 and cmsg_type equals either IPV6_HOPOPTS or IPV6_DSTOPTS.

tptrp is a pointer to a pointer to an 8-bit byte and ∗*tptrp* is used by the function to remember its place in the ancillary data object each time the function is called. The first time this function is called for a given ancillary data object, ∗*tptrp* must be set to NULL.

Each time this function returns success, **tptrp* points to the 8-bit option type field for the next option to be processed.

inet6_option_find

This function is similar to the previously described **inet6_option_next**() function, except this function lets the caller specify the option type to be searched for, instead of always returning the next option in the ancillary data object. *cmsg* is a pointer to `cmsghdr` structure of which `cmsg_level` equals `IPPROTO_IPV6` and `cmsg_type` equals either `IPV6_HOPOPTS` or `IPV6_DSTOPTS`.

tptrp is a pointer to a pointer to an 8-bit byte and **tptrp* is used by the function to remember its place in the ancillary data object each time the function is called. The first time this function is called for a given ancillary data object, **tptrp* must be set to `NULL`. ~ This function starts searching for an option of the specified type beginning after the value of **tptrp*. If an option of the specified type is located, this function returns `0` and **tptrp* points to the 8- bit option type field for the option of the specified type. If an option of the specified type is not located, the return value is `-1` and **tptrp* is `NULL`. If an error occurs, the return value is `-1` and **tptrp* is not `NULL`.

EXAMPLES

RFC 2292 gives comprehensive examples in chapter 6.

DIAGNOSTICS

inet6_option_init() and **inet6_option_append**() return `0` on success or `-1` on an error.

inet6_option_alloc() returns `NULL` on an error.

On errors, **inet6_option_next**() and **inet6_option_find**() return `-1` setting **tptrp* to non `NULL` value.

SEE ALSO

W. Stevens and M. Thomas, *Advanced Sockets API for IPv6*, RFC 2292, February 1998.

S. Deering and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 2460, December 1998.

STANDARDS

The functions are documented in “Advanced Sockets API for IPv6” (RFC 2292) .

HISTORY

The implementation first appeared in KAME advanced networking kit.

BUGS

The text was shamelessly copied from RFC 2292.

NAME

inet6_rth_space, **inet6_rth_init**, **inet6_rth_add**, **inet6_rth_reverse**, **inet6_rth_segments**, **inet6_rth_getaddr** — IPv6 Routing Header Options manipulation

SYNOPSIS

```
#include <netinet/in.h>

socklen_t
inet6_rth_space(int, int);

void *
inet6_rth_init(void *, socklen_t, int, int);

int
inet6_rth_add(void *, const struct in6_addr *);

int
inet6_rth_reverse(const void *, void *);

int
inet6_rth_segments(const void *);

struct in6_addr *
inet6_rth_getaddr(const void *, int);
```

DESCRIPTION

The IPv6 Advanced API, RFC 3542, defines the functions that an application calls to build and examine IPv6 Routing headers. Routing headers are used to perform source routing in IPv6 networks. The RFC uses the word “segments” to describe addresses and that is the term used here as well. All of the functions are defined in the <netinet/in.h> header file. The functions described in this manual page all operate on routing header structures which are defined in <netinet/ip6.h> but which should not need to be modified outside the use of this API. The size and shape of the route header structures may change, so using the APIs is a more portable, long term, solution.

The functions in the API are split into two groups, those that build a routing header and those that parse a received routing header. We will describe the builder functions followed by the parser functions.

inet6_rth_space

The **inet6_rth_space**() function returns the number of bytes required to hold a Routing Header of the type, specified in the *type* argument and containing the number of addresses specified in the *segments* argumment. When the type is IPV6_RTHDR_TYPE_0 the number of segments must be from 0 through 127. Routing headers of type IPV6_RTHDR_TYPE_2 contain only one segment, and are only used with Mobile IPv6. The return value from this function is the number of bytes required to store the routing header. If the value 0 is returned then either the route header type was not recognized or another error occurred.

inet6_rth_init

The **inet6_rth_init**() function initializes the pre-allocated buffer pointed to by *bp* to contain a routing header of the specified type The *bp_len* argument is used to verify that the buffer is large enough. The caller must allocate the buffer pointed to by bp. The necessary buffer size should be determined by calling **inet6_rth_space**() described in the previous sections.

The **inet6_rth_init**() function returns a pointer to *bp* on success and NULL when there is an error.

inet6_rth_add

The **inet6_rth_add**() function adds the IPv6 address pointed to by *addr* to the end of the routing header being constructed.

A successful addition results in the function returning 0, otherwise −1 is returned.

inet6_rth_reverse

The **inet6_rth_reverse**() function takes a routing header, pointed to by the argument *in*, and writes a new routing header into the argument pointed to by *out*. The routing header at that sends datagrams along the reverse of that route. Both arguments are allowed to point to the same buffer meaning that the reversal can occur in place.

The return value of the function is 0 on success, or −1 when there is an error.

The next set of functions operate on a routing header that the application wants to parse. In the usual case such a routing header is received from the network, although these functions can also be used with routing headers that the application itself created.

inet6_rth_segments

The **inet6_rth_segments**() function returns the number of segments contained in the routing header pointed to by *bp*. The return value is the number of segments contained in the routing header, or −1 if an error occurred. It is not an error for 0 to be returned as a routing header may contain 0 segments.

inet6_rth_getaddr

The **inet6_rth_getaddr**() function is used to retrieve a single address from a routing header. The *index* is the location in the routing header from which the application wants to retrieve an address. The *index* parameter must have a value between 0 and one less than the number of segments present in the routing header. The **inet6_rth_segments**() function, described in the last section, should be used to determine the total number of segments in the routing header. The **inet6_rth_getaddr**() function returns a pointer to an IPv6 address on success or NULL when an error has occurred.

DIAGNOSTICS

The **inet6_rth_space**() and **inet6_rth_getaddr**() functions return 0 on errors.

The **inet6_rthdr_init**() function returns NULL on error. The **inet6_rth_add**() and **inet6_rth_reverse**() functions return 0 on success, or −1 upon an error.

EXAMPLES

RFC 3542 gives extensive examples in Section 21, Appendix B.

KAME also provides examples in the advapitest directory of its kit.

SEE ALSO

W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei, *Advanced Sockets API for IPv6*, RFC 3542, May 2003.

S. Deering and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC2460, December 1998.

HISTORY

The implementation first appeared in KAME advanced networking kit.

NAME

inet6_rthdr_space, **inet6_rthdr_init**, **inet6_rthdr_add**, **inet6_rthdr_lasthop**, **inet6_rthdr_reverse**, **inet6_rthdr_segments**, **inet6_rthdr_getaddr**, **inet6_rthdr_getflags** — IPv6 Routing Header Options manipulation

SYNOPSIS

```
#include <netinet/in.h>

size_t
inet6_rthdr_space(int type, int segments);

struct cmsghdr *
inet6_rthdr_init(void *bp, int type);

int
inet6_rthdr_add(struct cmsghdr *cmsg, const struct in6_addr *addr,
    unsigned int flags);

int
inet6_rthdr_lasthop(struct cmsghdr *cmsg, unsigned int flags);

int
inet6_rthdr_reverse(const struct cmsghdr *in, struct cmsghdr *out);

int
inet6_rthdr_segments(const struct cmsghdr *cmsg);

struct in6_addr *
inet6_rthdr_getaddr(struct cmsghdr *cmsg, int index);

int
inet6_rthdr_getflags(const struct cmsghdr *cmsg, int index);
```

DESCRIPTION

RFC 2292 IPv6 advanced API defines eight functions that the application calls to build and examine a Routing header. Four functions build a Routing header:

inet6_rthdr_space() return #bytes required for ancillary data

inet6_rthdr_init() initialize ancillary data for Routing header

inet6_rthdr_add() add IPv6 address & flags to Routing header

inet6_rthdr_lasthop() specify the flags for the final hop

Four functions deal with a returned Routing header:

inet6_rthdr_reverse() reverse a Routing header

inet6_rthdr_segments() return #segments in a Routing header

inet6_rthdr_getaddr() fetch one address from a Routing header

inet6_rthdr_getflags() fetch one flag from a Routing header

The function prototypes for these functions are all in the ⟨`netinet/in.h`⟩ header.

inet6_rthdr_space

This function returns the number of bytes required to hold a Routing header of the specified *`type`* containing the specified number of *`segments`* (addresses). For an IPv6 Type 0 Routing header, the number of segments must be between 1 and 23, inclusive. The return value includes the size of the cmsghdr structure

that precedes the Routing header, and any required padding.

If the return value is 0, then either the type of the Routing header is not supported by this implementation or the number of segments is invalid for this type of Routing header.

Note: This function returns the size but does not allocate the space required for the ancillary data. This allows an application to allocate a larger buffer, if other ancillary data objects are desired, since all the ancillary data objects must be specified to `sendmsg`(2) as a single `msg_control` buffer.

inet6_rthdr_init

This function initializes the buffer pointed to by *`bp`* to contain a `cmsghdr` structure followed by a Routing header of the specified *`type`*. The `cmsg_len` member of the `cmsghdr` structure is initialized to the size of the structure plus the amount of space required by the Routing header. The `cmsg_level` and `cmsg_type` members are also initialized as required.

The caller must allocate the buffer and its size can be determined by calling **`inet6_rthdr_space`**().

Upon success the return value is the pointer to the `cmsghdr` structure, and this is then used as the first argument to the next two functions. Upon an error the return value is `NULL`.

inet6_rthdr_add

This function adds the address pointed to by *`addr`* to the end of the Routing header being constructed and sets the type of this hop to the value of *`flags`*. For an IPv6 Type 0 Routing header, *`flags`* must be either `IPV6_RTHDR_LOOSE` or `IPV6_RTHDR_STRICT`.

If successful, the `cmsg_len` member of the `cmsghdr` structure is updated to account for the new address in the Routing header and the return value of the function is 0. Upon an error the return value of the function is -1.

inet6_rthdr_lasthop

This function specifies the Strict/Loose flag for the final hop of a Routing header. For an IPv6 Type 0 Routing header, *`flags`* must be either `IPV6_RTHDR_LOOSE` or `IPV6_RTHDR_STRICT`.

The return value of the function is 0 upon success, or -1 upon an error.

Notice that a Routing header specifying `N` intermediate nodes requires `N+1` Strict/Loose flags. This requires `N` calls to **`inet6_rthdr_add`**() followed by one call to **`inet6_rthdr_lasthop`**().

inet6_rthdr_reverse

This function takes a Routing header that was received as ancillary data (pointed to by the first argument, *`in`*) and writes a new Routing header that sends datagrams along the reverse of that route. Both arguments are allowed to point to the same buffer (that is, the reversal can occur in place).

The return value of the function is 0 on success, or -1 upon an error.

inet6_rthdr_segments

This function returns the number of segments (addresses) contained in the Routing header described by *`cmsg`*. On success the return value is between 1 and 23, inclusive. The return value of the function is -1 upon an error.

inet6_rthdr_getaddr

This function returns a pointer to the IPv6 address specified by *`index`* (which must have a value between 1 and the value returned by **`inet6_rthdr_segments`**()) in the Routing header described by *`cmsg`*. An application should first call **`inet6_rthdr_segments`**() to obtain the number of segments in the Routing header.

Upon an error the return value of the function is `NULL`.

inet6_rthdr_getflags

This function returns the flags value specified by *index* (which must have a value between 0 and the value returned by **inet6_rthdr_segments**()) in the Routing header described by *cmsg*. For an IPv6 Type 0 Routing header the return value will be either `IPV6_RTHDR_LOOSE` or `IPV6_RTHDR_STRICT`.

Upon an error the return value of the function is -1.

Note: Addresses are indexed starting at 1, and flags starting at 0, to maintain consistency with the terminology and figures in RFC 2460.

EXAMPLES

RFC 2292 gives comprehensive examples in chapter 8.

DIAGNOSTICS

inet6_rthdr_space() returns 0 on errors.

inet6_rthdr_add(), **inet6_rthdr_lasthop**() and **inet6_rthdr_reverse**() return 0 on success, and returns -1 on error.

inet6_rthdr_init() and **inet6_rthdr_getaddr**() return `NULL` on error.

inet6_rthdr_segments() and **inet6_rthdr_getflags**() return -1 on error.

SEE ALSO

W. Stevens and M. Thomas, *Advanced Sockets API for IPv6*, RFC 2292, February 1998.

S. Deering and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 2460, December 1998.

STANDARDS

The functions are documented in "Advanced Sockets API for IPv6" (RFC 2292).

HISTORY

The implementation first appeared in KAME advanced networking kit.

BUGS

The text was shamelessly copied from RFC 2292.

inet6_rthdr_reverse() is not implemented yet.

NAME

inet6_rthdr_space, **inet6_rthdr_init**, **inet6_rthdr_add**, **inet6_rthdr_lasthop**, **inet6_rthdr_reverse**, **inet6_rthdr_segments**, **inet6_rthdr_getaddr**, **inet6_rthdr_getflags** — IPv6 Routing Header Options manipulation

SYNOPSIS

```
#include <netinet/in.h>

size_t
inet6_rthdr_space(int type, int segments);

struct cmsghdr *
inet6_rthdr_init(void *bp, int type);

int
inet6_rthdr_add(struct cmsghdr *cmsg, const struct in6_addr *addr,
    unsigned int flags);

int
inet6_rthdr_lasthop(struct cmsghdr *cmsg, unsigned int flags);

int
inet6_rthdr_reverse(const struct cmsghdr *in, struct cmsghdr *out);

int
inet6_rthdr_segments(const struct cmsghdr *cmsg);

struct in6_addr *
inet6_rthdr_getaddr(struct cmsghdr *cmsg, int index);

int
inet6_rthdr_getflags(const struct cmsghdr *cmsg, int index);
```

DESCRIPTION

RFC 2292 IPv6 advanced API defines eight functions that the application calls to build and examine a Routing header. Four functions build a Routing header:

inet6_rthdr_space() return #bytes required for ancillary data

inet6_rthdr_init() initialize ancillary data for Routing header

inet6_rthdr_add() add IPv6 address & flags to Routing header

inet6_rthdr_lasthop() specify the flags for the final hop

Four functions deal with a returned Routing header:

inet6_rthdr_reverse() reverse a Routing header

inet6_rthdr_segments() return #segments in a Routing header

inet6_rthdr_getaddr() fetch one address from a Routing header

inet6_rthdr_getflags() fetch one flag from a Routing header

The function prototypes for these functions are all in the ⟨`netinet/in.h`⟩ header.

inet6_rthdr_space

This function returns the number of bytes required to hold a Routing header of the specified *`type`* containing the specified number of *`segments`* (addresses). For an IPv6 Type 0 Routing header, the number of segments must be between 1 and 23, inclusive. The return value includes the size of the cmsghdr structure

that precedes the Routing header, and any required padding.

If the return value is 0, then either the type of the Routing header is not supported by this implementation or the number of segments is invalid for this type of Routing header.

Note: This function returns the size but does not allocate the space required for the ancillary data. This allows an application to allocate a larger buffer, if other ancillary data objects are desired, since all the ancillary data objects must be specified to `sendmsg`(2) as a single `msg_control` buffer.

inet6_rthdr_init

This function initializes the buffer pointed to by *`bp`* to contain a `cmsghdr` structure followed by a Routing header of the specified *`type`*. The `cmsg_len` member of the `cmsghdr` structure is initialized to the size of the structure plus the amount of space required by the Routing header. The `cmsg_level` and `cmsg_type` members are also initialized as required.

The caller must allocate the buffer and its size can be determined by calling **`inet6_rthdr_space`**().

Upon success the return value is the pointer to the `cmsghdr` structure, and this is then used as the first argument to the next two functions. Upon an error the return value is `NULL`.

inet6_rthdr_add

This function adds the address pointed to by *`addr`* to the end of the Routing header being constructed and sets the type of this hop to the value of *`flags`*. For an IPv6 Type 0 Routing header, *`flags`* must be either `IPV6_RTHDR_LOOSE` or `IPV6_RTHDR_STRICT`.

If successful, the `cmsg_len` member of the `cmsghdr` structure is updated to account for the new address in the Routing header and the return value of the function is 0. Upon an error the return value of the function is -1.

inet6_rthdr_lasthop

This function specifies the Strict/Loose flag for the final hop of a Routing header. For an IPv6 Type 0 Routing header, *`flags`* must be either `IPV6_RTHDR_LOOSE` or `IPV6_RTHDR_STRICT`.

The return value of the function is 0 upon success, or -1 upon an error.

Notice that a Routing header specifying `N` intermediate nodes requires `N+1` Strict/Loose flags. This requires `N` calls to **`inet6_rthdr_add`**() followed by one call to **`inet6_rthdr_lasthop`**().

inet6_rthdr_reverse

This function takes a Routing header that was received as ancillary data (pointed to by the first argument, *`in`*) and writes a new Routing header that sends datagrams along the reverse of that route. Both arguments are allowed to point to the same buffer (that is, the reversal can occur in place).

The return value of the function is 0 on success, or -1 upon an error.

inet6_rthdr_segments

This function returns the number of segments (addresses) contained in the Routing header described by *`cmsg`*. On success the return value is between 1 and 23, inclusive. The return value of the function is -1 upon an error.

inet6_rthdr_getaddr

This function returns a pointer to the IPv6 address specified by *`index`* (which must have a value between 1 and the value returned by **`inet6_rthdr_segments`**()) in the Routing header described by *`cmsg`*. An application should first call **`inet6_rthdr_segments`**() to obtain the number of segments in the Routing header.

Upon an error the return value of the function is `NULL`.

inet6_rthdr_getflags

This function returns the flags value specified by *index* (which must have a value between 0 and the value returned by **inet6_rthdr_segments**()) in the Routing header described by *cmsg*. For an IPv6 Type 0 Routing header the return value will be either `IPV6_RTHDR_LOOSE` or `IPV6_RTHDR_STRICT`.

Upon an error the return value of the function is -1.

Note: Addresses are indexed starting at 1, and flags starting at 0, to maintain consistency with the terminology and figures in RFC 2460.

EXAMPLES

RFC 2292 gives comprehensive examples in chapter 8.

DIAGNOSTICS

inet6_rthdr_space() returns 0 on errors.

inet6_rthdr_add(), **inet6_rthdr_lasthop**() and **inet6_rthdr_reverse**() return 0 on success, and returns -1 on error.

inet6_rthdr_init() and **inet6_rthdr_getaddr**() return `NULL` on error.

inet6_rthdr_segments() and **inet6_rthdr_getflags**() return -1 on error.

SEE ALSO

W. Stevens and M. Thomas, *Advanced Sockets API for IPv6*, RFC 2292, February 1998.

S. Deering and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 2460, December 1998.

STANDARDS

The functions are documented in "Advanced Sockets API for IPv6" (RFC 2292).

HISTORY

The implementation first appeared in KAME advanced networking kit.

BUGS

The text was shamelessly copied from RFC 2292.

inet6_rthdr_reverse() is not implemented yet.

NAME

inet_net_ntop, **inet_net_pton** — Internet network number manipulation routines

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *
inet_net_ntop(int af, const void *src, int bits, char *dst, size_t size);

int
inet_net_pton(int af, const char *src, void *dst, size_t size);
```

DESCRIPTION

The **inet_net_ntop**() function converts an Internet network number from network format (usually a *struct in_addr* or some other binary form, in network byte order) to CIDR presentation format (suitable for external display purposes). *bits* is the number of bits in *src* that are the network number. It returns NULL if a system error occurs (in which case, *errno* will have been set), or it returns a pointer to the destination string.

The **inet_net_pton**() function converts a presentation format Internet network number (that is, printable form as held in a character string) to network format (usually a *struct in_addr* or some other internal binary representation, in network byte order). It returns the number of bits (either computed based on the class, or specified with /CIDR), or -1 if a failure occurred (in which case *errno* will have been set. It will be set to ENOENT if the Internet network number was not valid).

The currently supported values for *af* are AF_INET and AF_INET6. *size* is the size of the result buffer *dst*.

NETWORK NUMBERS (IP VERSION 4)

Internet network numbers may be specified in one of the following forms:

```
a.b.c.d/bits
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet network number. Note that when an Internet network number is viewed as a 32-bit integer quantity on a system that uses little-endian byte order (such as the Intel 386, 486 and Pentium processors) the bytes referred to above appear as "d.c.b.a". That is, little-endian bytes are ordered from right to left.

When a three part number is specified, the last part is interpreted as a 16-bit quantity and placed in the rightmost two bytes of the Internet network number. This makes the three part number format convenient for specifying Class B network numbers as "128.net.host".

When a two part number is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the Internet network number. This makes the two part number format convenient for specifying Class A network numbers as "net.host".

When only one part is given, the value is stored directly in the Internet network number without any byte rearrangement.

All numbers supplied as "parts" in a '.' notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

SEE ALSO

byteorder(3), inet(3), networks(5)

HISTORY

The **inet_net_ntop** and **inet_net_pton** functions appeared in BIND 4.9.4 and thence NetBSD 1.3. Support for AF_INET6 appeared in NetBSD 1.6.

NAME

initgroups — initialize supplementary group IDs

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
initgroups(const char *name, gid_t basegid);
```

DESCRIPTION

The **initgroups**() function uses the getgrouplist(3) function to calculate the supplementary group IDs for the user specified in *name*. This group list is then set up for the current process using setgroups(2). The *basegid* is automatically included in the group list. Typically this value is given as the group number from the password file.

If the groups database lists more than NGROUPS groups for *name* (including one for *basegid*), the later groups are ignored.

RETURN VALUES

The **initgroups**() function returns −1 if it was not invoked by the super-user.

SEE ALSO

setgroups(2), getgrouplist(3)

HISTORY

The **initgroups**() function appeared in 4.2BSD.

BUGS

The **getgrouplist**() function called by **initgroups**() uses the routines based on getgrent(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to **initgroups**().

NAME

insque, **remque** — insert/remove element from a queue

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <search.h>

void
insque(void *elem, void *pred);

void
remque(void *elem);
```

DESCRIPTION

insque() and **remque**() manipulate queues built from doubly linked lists. The queue can be either circular or linear. The functions expect their arguments to point to a structure whose first and second members are pointers to the next and previous element, respectively. The **insque**() function also allows the *pred* argument to be a NULL pointer for the initialization of a new linear list's head element.

STANDARDS

The **insque**() and **remque**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

These are derived from the insque and remque instructions on a VAX.

NAME

ipsec_set_policy, **ipsec_get_policylen**, **ipsec_dump_policy** — manipulate IPsec policy specification structure from human-readable policy string

LIBRARY

IPsec Policy Control Library (libipsec, −lipsec)

SYNOPSIS

```
#include <netinet6/ipsec.h>

char *
ipsec_set_policy(char *policy, int len);

int
ipsec_get_policylen(char *buf);

char *
ipsec_dump_policy(char *buf, char *delim);
```

DESCRIPTION

ipsec_set_policy() generates an IPsec policy specification structure, namely `struct sadb_x_policy` and/or `struct sadb_x_ipsecrequest` from a human-readable policy specification. The policy specification must be given as a C string *policy* and its length *len*. **ipsec_set_policy**() will return a buffer with the corresponding IPsec policy specification structure. The buffer is dynamically allocated, and must be `free`(3)'d by the caller.

You can get the length of the generated buffer with **ipsec_get_policylen**() (i.e. for calling `setsockopt`(2)).

ipsec_dump_policy() converts an IPsec policy structure into human-readable form. Therefore, **ipsec_dump_policy**() can be regarded as the inverse function to **ipsec_set_policy**(). *buf* points to an IPsec policy structure, `struct sadb_x_policy`. *delim* is a delimiter string, which is usually a blank character. If you set *delim* to `NULL`, a single whitespace is assumed. **ipsec_dump_policy**() returns a pointer to a dynamically allocated string. It is the caller's responsibility to `free`(3) it.

policy is formatted as either of the following:

direction [priority specification] `discard`

> *direction* must be `in`, `out`, or `fwd`. *direction* specifies in which direction the policy needs to be applied. The non-standard direction `fwd` is substituted with `in` on platforms which do not support forward policies.
>
> *priority specification* is used to control the placement of the policy within the SPD. The policy position is determined by a signed integer where higher priorities indicate the policy is placed closer to the beginning of the list and lower priorities indicate the policy is placed closer to the end of the list. Policies with equal priorities are added at the end of the group of such policies.
>
> Priority can only be specified when libipsec has been compiled against kernel headers that support policy priorities (Linux >= 2.6.6). It takes one of the following formats:
>
> *{priority,prio} offset*
>
> > *offset* is an integer in the range -2147483647..214783648.
>
> *{priority,prio} base {+,-} offset*
>
> > *base* is either `low (-1073741824)`, `def (0)`, or `high (1073741824)`.

offset is an unsigned integer. It can be up to 1073741824 for positive offsets, and up to 1073741823 for negative offsets.

The interpretation of policy priority in these functions and the kernel DOES differ. The relationship between the two can be described as p(kernel) = 0x80000000 - p(func)

With discard policy, packets will be dropped if they match the policy.

direction [priority specification] entrust
entrust means to consult the SPD defined by setkey(8).

direction [priority specification] bypass
bypass means to bypass the IPsec processing. (the packet will be transmitted in clear). This is for privileged sockets.

direction [*priority specification*] ipsec *request ...*
ipsec means that the matching packets are subject to IPsec processing. ipsec can be followed by one or more *request* strings, which are formatted as below:

protocol / *mode* / *src* - *dst* [/*level*]
protocol is either ah, esp, or ipcomp.

mode is either transport or tunnel.

src and *dst* specifies the IPsec endpoint. *src* always means the "sending node" and *dst* always means the "receiving node". Therefore, when *direction* is in, *dst* is this node and *src* is the other node (peer). If *mode* is transport, Both *src* and *dst* can be omitted.

level must be set to one of the following: default, use, require, or unique. default means that the kernel should consult the system default policy defined by sysctl(8), such as net.inet.ipsec.esp_trans_deflev. See ipsec(4) regarding the system default. use means that a relevant SA can be used when available, since the kernel may perform IPsec operation against packets when possible. In this case, packets can be transmitted in clear (when SA is not available), or encrypted (when SA is available). require means that a relevant SA is required, since the kernel must perform IPsec operation against packets. unique is the same as require, but adds the restriction that the SA for outbound traffic is used only for this policy. You may need the identifier in order to relate the policy and the SA when you define the SA by manual keying. You can put the decimal number as the identifier after unique like unique: number. number must be between 1 and 32767 . If the *request* string is kept unambiguous, *level* and slash prior to *level* can be omitted. However, it is encouraged to specify them explicitly to avoid unintended behavior. If *level* is omitted, it will be interpreted as default.

Note that there are slight differences to the specification of setkey(8). In the specification of setkey(8), both entrust and bypass are not used. Refer to setkey(8) for details.

Here are several examples (long lines are wrapped for readability):

```
in discard
out ipsec esp/transport//require
in ipsec ah/transport//require
out ipsec esp/tunnel/10.1.1.2-10.1.1.1/use
in ipsec ipcomp/transport//use
        esp/transport//use
```

RETURN VALUES

ipsec_set_policy() returns a pointer to the allocated buffer with the policy specification if successful; otherwise a NULL pointer is returned. **ipsec_get_policylen**() returns a positive value (meaning the buffer size) on success, and a negative value on errors. **ipsec_dump_policy**() returns a pointer to a dynamically allocated region on success, and NULL on errors.

SEE ALSO

ipsec_strerror(3), ipsec(4), setkey(8)

HISTORY

The functions first appeared in the WIDE/KAME IPv6 protocol stack kit.

NAME

ipsec_strerror — error messages for the IPsec policy manipulation library

LIBRARY

IPsec Policy Control Library (libipsec, −lipsec)

SYNOPSIS

```
#include <netinet6/ipsec.h>

const char *
ipsec_strerror(void);
```

DESCRIPTION

`netinet6/ipsec.h` declares

```
extern int ipsec_errcode;
```

which is used to pass an error code from the IPsec policy manipulation library to a program. **ipsec_strerror**() can be used to obtain the error message string for the error code.

The array pointed to is not to be modified by the calling program. Since **ipsec_strerror**() uses `strerror`(3) as underlying function, calling `strerror`(3) after **ipsec_strerror**() will make the return value from **ipsec_strerror**() invalid or overwritten.

RETURN VALUES

ipsec_strerror() always returns a pointer to a C string. The C string must not be overwritten by the calling program.

SEE ALSO

`ipsec_set_policy`(3)

HISTORY

ipsec_strerror() first appeared in the WIDE/KAME IPv6 protocol stack kit.

BUGS

ipsec_strerror() will return its result which may be overwritten by subsequent calls.

ipsec_errcode is not thread safe.

NAME

isalnum — alphanumeric character test

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
isalnum(int c);
```

DESCRIPTION

The **isalnum**() function tests for any character for which isalpha(3) or isdigit(3) is true.

RETURN VALUES

The **isalnum**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **isalnum**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

CAVEATS

The argument to **isalnum**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

isalpha — alphabetic character test

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
isalpha(int c);
```

DESCRIPTION

The **isalpha**() function tests for any character for which isupper(3) or islower(3) is true and for which none of iscntrl(3), isdigit(3), ispunct(3), or isspace(3) is true. In the *"C"* locale, **isalpha**() returns true only for the characters for which isupper(3) or islower(3) is true.

RETURN VALUES

The **isalpha**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalnum(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **isalpha**() function conforms to ANSI X3.159-1989 ("ANSI C89").

CAVEATS

The argument to **isalpha**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

isascii — test for ASCII character

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
isascii(int c);
```

DESCRIPTION

The **isascii**() function tests for an ASCII character, which is any character with a value in the range from 0 to 127, inclusive.

The **isascii**() is defined on all integer values.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **isascii**() function conforms to X/Open Portability Guide Issue 4 (“XPG4”).

NAME

isblank — blank-space character test

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
isblank(int c);
```

DESCRIPTION

The **isblank**() function tests for the standard blank-space characters. The standard blank-space characters are the following:

‘ ’	Space character.
\t	Horizontal tab.

In the *“C”* locale, **isblank**() returns true only for the standard blank-space characters.

RETURN VALUES

The **isblank**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

CAVEATS

The argument to **isblank**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

iscntrl — control character test

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
iscntrl(int c);
```

DESCRIPTION

The **iscntrl**() function tests for any control character.

RETURN VALUES

The **iscntrl**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), isblank(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **iscntrl**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

CAVEATS

The argument to **iscntrl**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

isdigit — decimal-digit character test

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
isdigit(int c);
```

DESCRIPTION

The **isdigit**() function tests for any decimal-digit character.

RETURN VALUES

The **isdigit**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **isdigit**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

CAVEATS

The argument to **isdigit**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

isfinite — test for finite value

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <math.h>

int
isfinite(real-floating x);
```

DESCRIPTION

The **isfinite**() determines whether its argument *x* has a finite value. An argument represented in a format wider than its semantic type is converted to its semantic type first. The determination is then based on the type of the argument.

IEEE 754

It is determined whether the value of *x* is zero, subnormal, or normal, and neither infinite nor NaN.

VAX

It is determined whether the value of *x* is true zero or finite, and neither dirty zero nor ROP.

RETURN VALUES

The **isfinite**() macro returns a non-zero value if the value of *x* is finite. Otherwise 0 is returned.

ERRORS

No errors are defined.

SEE ALSO

fpclassify(3), isnormal(3), math(3), signbit(3)

STANDARDS

The **isfinite**() macro conforms to ISO/IEC 9899:1999 (“ISO C99”).

NAME

isgraph — printing character test (space character exclusive)

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
isgraph(int c);
```

DESCRIPTION

The **isgraph**() function tests for any printing character except space (' ').

RETURN VALUES

The **isgraph**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **isgraph**() function conforms to ANSI X3.159-1989 ("ANSI C89").

CAVEATS

The argument to **isgraph**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

isgreater, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, **isunordered** — compare two floating-point numbers

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <math.h>

int
isgreater(real-floating x, real-floating y);

int
isgreaterequal(real-floating x, real-floating y);

int
isless(real-floating x, real-floating y);

int
islessequal(real-floating x, real-floating y);

int
islessgreater(real-floating x, real-floating y);

int
isunordered(real-floating x, real-floating y);
```

DESCRIPTION

Each of the macros **isgreater**(), **isgreaterequal**(), **isless**(), **islessequal**(), and **islessgreater**() take arguments *x* and *y* and return a non-zero value if and only if its nominal relation on *x* and *y* is true. These macros always return zero if either argument is not a number (NaN), but unlike the corresponding C operators, they never raise a floating point exception.

The **isunordered**() macro takes arguments *x* and *y* and returns non-zero if and only if neither *x* nor *y* are NaNs. For any pair of floating-point values, one of the relationships (less, greater, equal, unordered) holds.

SEE ALSO

fpclassify(3), math(3), signbit(3)

STANDARDS

The **isgreater**(), **isgreaterequal**(), **isless**(), **islessequal**(), **islessgreater**(), and **isunordered**() macros conform to ISO/IEC 9899:1999 (“ISO C99”).

HISTORY

The relational macros described above first appeared in NetBSD 5.0.

NAME

isinf — test for infinity

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <math.h>

int
isinf(real-floating x);
```

DESCRIPTION

The **isinf**() macro determines whether its argument *x* is an infinity (positive or negative). An argument represented in a format wider than its semantic type is converted to its semantic type first. The determination is then based on the type of the argument.

IEEE 754

It is determined whether the value of *x* is an infinity.

VAX

Infinities are not supported.

RETURN VALUES

The **isinf**() macro returns a non-zero value if the value of *x* is an infinity. Otherwise 0 is returned.

SEE ALSO

fpclassify(3), isfinite(3), isinff(3), isnan(3), isnanf(3), isnormal(3), math(3), signbit(3)

IEEE Standard for Binary Floating-Point Arithmetic, Std 754-1985, ANSI.

STANDARDS

The **isinf**() macro conforms to ISO/IEC 9899:1999 (“ISO C99”).

NAME

isinff, **isnanf** — test for infinity or not-a-number

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

int
isinff(float);

int
isnanf(float);
```

DESCRIPTION

The **isinff**() function returns 1 if the number is “∞”, otherwise 0.

The **isnanf**() function returns 1 if the number is “not-a-number”, otherwise 0.

SEE ALSO

isinf(3), isnan(3), math(3)

IEEE Standard for Binary Floating-Point Arithmetic, Std 754-1985, ANSI.

BUGS

Neither the VAX nor the Tahoe floating point have distinguished values for either infinity or not-a-number. These routines always return 0 on those architectures.

NAME

islower — lower-case character test

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
islower(int c);
```

DESCRIPTION

The **islower**() function tests for any lower-case letter for which none of iscntrl(3), isdigit(3), ispunct(3), or isspace(3) is true. In the *"C"* locale, **islower**() returns true only for the characters defined as lower-case letters.

RETURN VALUES

The **islower**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **islower**() function conforms to ANSI X3.159-1989 ("ANSI C89").

CAVEATS

The argument to **islower**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

isnan — test for not-a-number

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <math.h>

int
isnan(real-floating x);
```

DESCRIPTION

The **isnan**() macro determines whether its argument *x* is not-a-number ("NaN"). An argument represented in a format wider than its semantic type is converted to its semantic type first. The determination is then based on the type of the argument.

IEEE 754

It is determined whether the value of *x* is a NaN.

VAX

NaNs are not supported.

RETURN VALUES

The **isnan**() macro returns a non-zero value if the value of *x* is a NaN. Otherwise 0 is returned.

SEE ALSO

fpclassify(3), isfinite(3), isinf(3), isinff(3), isnanf(3), isnormal(3), math(3), signbit(3)

IEEE Standard for Binary Floating-Point Arithmetic, Std 754-1985, ANSI.

STANDARDS

The **isnan**() macro conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

isnormal — test for normal value

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <math.h>

int
isnormal(real-floating x);
```

DESCRIPTION

The **isnormal**() macro determines whether its argument *x* has a normal value. An argument represented in a format wider than its semantic type is converted to its semantic type first. The determination is then based on the type of the argument.

IEEE 754

It is determined whether the value of *x* is normal, and neither zero, subnormal, infinite nor NaN.

VAX

It is determined whether the value of *x* is finite, and neither true zero, dirty zero nor ROP.

RETURN VALUES

The **isnormal**() macro returns a non-zero value if the value of *x* is finite. Otherwise 0 is returned.

ERRORS

No errors are defined.

SEE ALSO

fpclassify(3), isfinite(3), math(3), signbit(3)

STANDARDS

The **isnormal**() macro conforms to ISO/IEC 9899:1999 (“ISO C99”).

NAME

iso_addr, **iso_ntoa** — elementary network address conversion routines for Open System Interconnection

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <netiso/iso.h>

struct iso_addr *
iso_addr(const char *cp);

char *
iso_ntoa(struct iso_addr *isoa);
```

DESCRIPTION

The routine **iso_addr**() interprets character strings representing OSI addresses, returning binary information suitable for use in system calls. The routine **iso_ntoa**() takes OSI addresses and returns ASCII strings representing NSAPs (network service access points) in a notation inverse to that accepted by **iso_addr**().

Unfortunately, no universal standard exists for representing OSI network addresses.

The format employed by **iso_addr**() is a sequence of hexadecimal "digits" (optionally separated by periods), of the form:

<hex digits>.<hex digits>.<hex digits>

Each pair of hexadecimal digits represents a byte with the leading digit indicating the higher-ordered bits. A period following an even number of bytes has no effect (but may be used to increase legibility). A period following an odd number of bytes has the effect of causing the byte of address being translated to have its higher order bits filled with zeros.

RETURN VALUES

iso_ntoa() always returns a null terminated string. **iso_addr**() always returns a pointer to a struct iso_addr. (See **BUGS**.)

SEE ALSO

iso(4)

HISTORY

The **iso_addr**() and **iso_ntoa**() functions appeared in 4.3BSD−Reno.

BUGS

The returned values reside in a static memory area.

The function **iso_addr**() should diagnose improperly formed input, and there should be an unambiguous way to recognize this.

NAME

isprint — printing character test (space character inclusive)

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
isprint(int c);
```

DESCRIPTION

The **isprint**() function tests for any printing character including space (' ').

RETURN VALUES

The **isprint**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **isprint**() function conforms to ANSI X3.159-1989 ("ANSI C89").

CAVEATS

The argument to **isprint**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

ispunct — punctuation character test

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
ispunct(int c);
```

DESCRIPTION

The **ispunct**() function tests for any printing character except space (' ') or a character for which isalnum(3) is true.

RETURN VALUES

The **ispunct**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **ispunct**() function conforms to ANSI X3.159-1989 ("ANSI C89").

CAVEATS

The argument to **ispunct**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

isspace — white-space character test

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
isspace(int c);
```

DESCRIPTION

The **isspace**() function tests for the standard white-space characters for which isalnum(3) is false. The standard white-space characters are the following:

‘ ’	Space character.
\f	Form feed.
\n	New-line.
\r	Carriage return.
\t	Horizontal tab.
\v	And vertical tab.

In the *“C”* locale, **isspace**() returns true only for the standard white-space characters.

RETURN VALUES

The **isspace**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **isspace**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

CAVEATS

The argument to **isspace**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

isupper — upper-case character test

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
isupper(int c);
```

DESCRIPTION

The **isupper**() function tests for any upper-case letter or any of an implementation-defined set of characters for which none of iscntrl(3), isdigit(3), ispunct(3), or isspace(3) is true. In the *"C"* locale, **isupper**() returns true only for the characters defined as upper-case letters.

RETURN VALUES

The **isupper**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isxdigit(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **isupper**() function conforms to ANSI X3.159-1989 ("ANSI C89").

CAVEATS

The argument to **isupper**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

iswalnum, **iswalpha**, **iswblank**, **iswcntrl**, **iswdigit**, **iswgraph**, **iswlower**, **iswprint**, **iswpunct**, **iswspace**, **iswupper**, **iswxdigit** — wide character classification utilities

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wctype.h>

int
iswalnum(wint_t wc);

int
iswalpha(wint_t wc);

int
iswblank(wint_t wc);

int
iswcntrl(wint_t wc);

int
iswdigit(wint_t wc);

int
iswgraph(wint_t wc);

int
iswlower(wint_t wc);

int
iswprint(wint_t wc);

int
iswpunct(wint_t wc);

int
iswspace(wint_t wc);

int
iswupper(wint_t wc);

int
iswxdigit(wint_t wc);
```

DESCRIPTION

The functions are character classification utility functions, for use with wide characters (*wchar_t* or *wint_t*). See the description of singlebyte classification functions, like isalnum(3), for details.

RETURN VALUES

The functions return zero if the character tests false and return non-zero if the character tests true.

SEE ALSO

isalnum(3), isalpha(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3)

STANDARDS

The functions conform to ISO/IEC 9899:1999 (“ISO C99”).

CAVEATS

The argument to these functions must be `WEOF` or a valid *`wchar_t`* value with the current locale; otherwise, the result is undefined.

NAME

iswctype — test a character for character class identifier

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wctype.h>

int
iswctype(wint_t wc, wctype_t charclass);
```

DESCRIPTION

The **iswctype**() function returns a boolean value that indicates whether a wide character *wc* is in *charclass*.

The behaviour of **iswctype**() is undefined if the **iswctype**() function is called with an invalid *charclass* (changes of LC_CTYPE category invalidate *charclass*) or invalid wide character *wc*.

The behaviour of **iswctype**() is affected by the LC_CTYPE category of the current locale.

RETURN VALUES

The **iswctype**() returns:

0	*wc* is not in *charclass*.
non-zero	*wc* is in *charclass*.

ERRORS

No errors are defined.

SEE ALSO

setlocale(3), towctrans(3), wctrans(3), wctype(3)

STANDARDS

The **iswctype**() function conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”).

NAME

isxdigit — hexadecimal-digit character test

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
isxdigit(int c);
```

DESCRIPTION

The **isxdigit**() function tests for any hexadecimal-digit character.

RETURN VALUES

The **isxdigit**() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), stdio(3), toascii(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **isxdigit**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

CAVEATS

The argument to **isxdigit**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

j0, **j0f**, **j1**, **j1f**, **jn**, **jnf**, **y0**, **y0f**, **y1**, **y1f**, **yn**, **ynf** — Bessel functions of first and second kind

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
j0(double x);

float
j0f(float x);

double
j1(double x);

float
j1f(float x);

double
jn(int n, double x);

float
jnf(int n, float x);

double
y0(double x);

float
y0f(float x);

double
y1(double x);

float
y1f(float x);

double
yn(int n, double x);

float
ynf(int n, float x);
```

DESCRIPTION

The functions **j0**(), **j0f**(), **j1**() and **j1f**() compute the *Bessel function of the first kind of the order* 0 and the *order* 1, respectively, for the real value *x*; the functions **jn**() and **jnf**() compute the *Bessel function of the first kind of the integer order n* for the real value *x*.

The functions **y0**(), **y0f**(), **y1**() and **y1f**() compute the linearly independent *Bessel function of the second kind of the order* 0 and the *order* 1, respectively, for the positive *integer* value *x* (expressed as a double); the functions **yn**() and **ynf**() compute the *Bessel function of the second kind for the integer order n* for the positive *integer* value *x* (expressed as a double).

RETURN VALUES

If these functions are successful, the computed value is returned, otherwise the global variable *errno* is set to `EDOM` and a reserve operand fault is generated.

SEE ALSO

`math`(3)

HISTORY

This set of functions appeared in Version 7 AT&T UNIX.

NAME

krb5_pwcheck, **kadm5_setup_passwd_quality_check**, **kadm5_add_passwd_quality_verifier**, **kadm5_check_password_quality** — Heimdal warning and error functions

LIBRARY

Kerberos 5 Library (libkadm5srv, -lkadm5srv)

SYNOPSIS

```
#include <kadm5-protos.h>
#include <kadm5-pwcheck.h>

void
kadm5_setup_passwd_quality_check(krb5_context context,
    const char *check_library, const char *check_function);

krb5_error_code
kadm5_add_passwd_quality_verifier(krb5_context context,
    const char *check_library);

const char *
kadm5_check_password_quality(krb5_context context,
    krb5_principal principal, krb5_data *pwd_data);

int
(*kadm5_passwd_quality_check_func)(krb5_context context,
    krb5_principal principal, krb5_data *password, const char *tuning,
    char *message, size_t length);
```

DESCRIPTION

These functions perform the quality check for the heimdal database library.

There are two versions of the shared object API; the old version (0) is deprecated, but still supported. The new version (1) supports multiple password quality checking modules in the same shared object. See below for details.

The password quality checker will run over all tests that are configured by the user.

Module names are of the form `vendor:test-name` or, if the the test name is unique enough, just `test-name`.

IMPLEMENTING A PASSWORD QUALITY CHECKING SHARED OBJECT

(This refers to the version 1 API only.)

Module shared objects may conveniently be compiled and linked with `libtool`(1). An object needs to export a symbol called `kadm5_password_verifier` of the type *`struct kadm5_pw_policy_verifier`*.

Its *`name`* and *`vendor`* fields should be contain the obvious information and *`version`* should be `KADM5_PASSWD_VERSION_V1`. *`funcs`* contains an array of *`struct kadm5_pw_policy_check_func`* structures that is terminated with an entry whose *`name`* component is `NULL`. The *`func`* Fields of the array elements are functions that are exported by the module to be called to check the password. They get the following arguments: the Kerberos context, principal, password, a tuning parameter, and a pointer to a message buffer and its length. The tuning parameter for the quality check function is currently always `NULL`. If the password is acceptable, the function returns zero. Otherwise it returns non-zero and fills in the message buffer with an appropriate explanation.

RUNNING THE CHECKS

kadm5_setup_passwd_quality_check sets up type 0 checks. It sets up all type 0 checks defined in krb5.conf(5) if called with the last two arguments null.

kadm5_add_passwd_quality_verifier sets up type 1 checks. It sets up all type 1 tests defined in krb5.conf(5) if called with a null second argument. **kadm5_check_password_quality** runs the checks in the order in which they are defined in krb5.conf(5) and the order in which they occur in a module's *funcs* array until one returns non-zero.

SEE ALSO

libtool(1), krb5(3), krb5.conf(5)

NAME

k_hasafs, **k_hasafs_recheck**, **k_pioctl**, **k_unlog**, **k_setpag**, **k_afs_cell_of_file**, **kafs_set_verbose**, **kafs_settoken_rxkad**, **kafs_settoken**, **krb_afslog**, **krb_afslog_uid**, **kafs_settoken5**, **krb5_afslog**, **krb5_afslog_uid** — AFS library

LIBRARY

AFS cache manager access library (libkafs, -lkafs)

SYNOPSIS

```
#include <kafs.h>

int
k_afs_cell_of_file(const char *path, char *cell, int len);

int
k_hasafs(void);

int
k_hasafs_recheck(void);

int
k_pioctl(char *a_path, int o_opcode, struct ViceIoctl *a_paramsP,
    int a_followSymlinks);

int
k_setpag(void);

int
k_unlog(void);

void
kafs_set_verbose(void (*func)(void *, const char *, int), void *);

int
kafs_settoken_rxkad(const char *cell, struct ClearToken *token,
    void *ticket, size_t ticket_len);

int
kafs_settoken(const char *cell, uid_t uid, CREDENTIALS *c);

krb_afslog(char *cell, char *realm);

int
krb_afslog_uid(char *cell, char *realm, uid_t uid);

krb5_error_code
krb5_afslog_uid(krb5_context context, krb5_ccache id, const char *cell,
    krb5_const_realm realm, uid_t uid);

int
kafs_settoken5(const char *cell, uid_t uid, krb5_creds *c);

krb5_error_code
krb5_afslog(krb5_context context, krb5_ccache id, const char *cell,
    krb5_const_realm realm);
```

DESCRIPTION

k_hasafs() initializes some library internal structures, and tests for the presence of AFS in the kernel, none of the other functions should be called before **k_hasafs**() is called, or if it fails.

k_hasafs_recheck() forces a recheck if a AFS client has started since last time **k_hasafs**() or **k_hasafs_recheck**() was called.

kafs_set_verbose() set a log function that will be called each time the kafs library does something important so that the application using libkafs can output verbose logging. Calling the function *kafs_set_verbose* with the function argument set to NULL will stop libkafs from calling the logging function (if set).

kafs_settoken_rxkad() set rxkad with the *token* and *ticket* (that have the length *ticket_len*) for a given *cell*.

kafs_settoken() and **kafs_settoken5**() work the same way as **kafs_settoken_rxkad**() but internally converts the Kerberos 4 or 5 credential to a afs cleartoken and ticket.

krb_afslog(), and **krb_afslog_uid**() obtains new tokens (and possibly tickets) for the specified *cell* and *realm*. If *cell* is NULL, the local cell is used. If *realm* is NULL, the function tries to guess what realm to use. Unless you have some good knowledge of what cell or realm to use, you should pass NULL. **krb_afslog**() will use the real user-id for the ViceId field in the token, **krb_afslog_uid**() will use *uid*.

krb5_afslog(), and **krb5_afslog_uid**() are the Kerberos 5 equivalents of **krb_afslog**(), and **krb_afslog_uid**().

krb5_afslog(), **kafs_settoken5**() can be configured to behave differently via a **krb5_appdefault** option afs-use-524 in krb5.conf. Possible values for afs-use-524 are:

yes use the 524 server in the realm to convert the ticket

no use the Kerberos 5 ticket directly, can be used with if the afs cell support 2b token.

local, 2b
convert the Kerberos 5 credential to a 2b token locally (the same work as a 2b 524 server should have done).

Example:

```
[appdefaults]
        SU.SE = { afs-use-524 = local }
        PDC.KTH.SE = { afs-use-524 = yes }
        afs-use-524 = yes
```

libkafs will use the libkafs as application name when running the **krb5_appdefault** function call.

The (uppercased) cell name is used as the realm to the **krb5_appdefault function.**

k_afs_cell_of_file() will in *cell* return the cell of a specified file, no more than *len* characters is put in *cell*.

k_pioctl() does a **pioctl**() system call with the specified arguments. This function is equivalent to **lpioctl**().

k_setpag() initializes a new PAG.

k_unlog() removes destroys all tokens in the current PAG.

RETURN VALUES

k_hasafs() returns 1 if AFS is present in the kernel, 0 otherwise. **krb_afslog**() and **krb_afslog_uid**() returns 0 on success, or a Kerberos error number on failure. **k_afs_cell_of_file**(), **k_pioctl**(), **k_setpag**(), and **k_unlog**() all return the value of the underlaying system call, 0 on success.

ENVIRONMENT

The following environment variable affect the mode of operation of **kafs**:

`AFS_SYSCALL` Normally, **kafs** will try to figure out the correct system call(s) that are used by AFS by itself. If it does not manage to do that, or does it incorrectly, you can set this variable to the system call number or list of system call numbers that should be used.

EXAMPLES

The following code from **login** will obtain a new PAG and tokens for the local cell and the cell of the users home directory.

```
if (k_hasafs()) {
        char cell[64];
        k_setpag();
        if(k_afs_cell_of_file(pwd->pw_dir, cell, sizeof(cell)) == 0)
                krb_afslog(cell, NULL);
        krb_afslog(NULL, NULL);
}
```

ERRORS

If any of these functions (apart from **k_hasafs**()) is called without AFS being present in the kernel, the process will usually (depending on the operating system) receive a SIGSYS signal.

SEE ALSO

`krb5_appdefault`(3), `krb5.conf`(5)

Transarc Corporation, "File Server/Cache Manager Interface", *AFS-3 Programmer's Reference*, 1991.

BUGS

`AFS_SYSCALL` has no effect under AIX.

NAME

killpg — send signal to a process group

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

int
killpg(pid_t pgrp, int sig);
```

DESCRIPTION

killpg() sends the signal *sig* to the process group *pgrp*. See sigaction(2) for a list of signals. If *pgrp* is 0, **killpg**() sends the signal to the sending process's process group.

The sending process and members of the process group must have the same effective user ID, or the sender must be the super-user. As a single special case the continue signal SIGCONT may be sent to any process that is a descendant of the current process.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

killpg() will fail and no signal will be sent if:

[EINVAL]	*sig* is not a valid signal number.
[ESRCH]	No process can be found in the process group specified by *pgrp*.
[ESRCH]	The process group was given as 0 but the sending process does not have a process group.
[EPERM]	The sending process is not the super-user and one or more of the target processes has an effective user ID different from that of the sending process.

SEE ALSO

getpgrp(2), kill(2), sigaction(2)

HISTORY

The **killpg**() function call appeared in 4.0BSD.

NAME

krb5 — Kerberos 5 library

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>
```

DESCRIPTION

These functions constitute the Kerberos 5 library, *libkrb5*.

LIST OF FUNCTIONS

Name/Page	*Description*

krb524_convert_creds_kdc.3
krb524_convert_creds_kdc_cache.3
krb5_425_conv_principal.3
krb5_425_conv_principal_ext.3
krb5_524_conv_principal.3
krb5_abort.3
krb5_abortx.3
krb5_acl_match_file.3
krb5_acl_match_string.3
krb5_add_et_list.3
krb5_add_extra_addresses.3
krb5_add_ignore_addresses.3
krb5_addlog_dest.3
krb5_addlog_func.3
krb5_addr2sockaddr.3
krb5_address.3
krb5_address_compare.3
krb5_address_order.3
krb5_address_search.3
krb5_addresses.3
krb5_aname_to_localname.3
krb5_anyaddr.3
krb5_appdefault_boolean.3
krb5_appdefault_string.3
krb5_appdefault_time.3
krb5_append_addresses.3
krb5_auth_con_addflags.3
krb5_auth_con_free.3
krb5_auth_con_genaddrs.3
krb5_auth_con_generatelocalsubkey.3
krb5_auth_con_getaddrs.3
krb5_auth_con_getauthenticator.3
krb5_auth_con_getcksumtype.3
krb5_auth_con_getflags.3
krb5_auth_con_getkey.3
krb5_auth_con_getkeytype.3
krb5_auth_con_getlocalseqnumber.3
krb5_auth_con_getlocalsubkey.3

krb5_auth_con_getrcache.3
krb5_auth_con_getremotesubkey.3
krb5_auth_con_getuserkey.3
krb5_auth_con_init.3
krb5_auth_con_initivector.3
krb5_auth_con_removeflags.3
krb5_auth_con_setaddrs.3
krb5_auth_con_setaddrs_from_fd.3
krb5_auth_con_setcksumtype.3
krb5_auth_con_setflags.3
krb5_auth_con_setivector.3
krb5_auth_con_setkey.3
krb5_auth_con_setkeytype.3
krb5_auth_con_setlocalseqnumber.3
krb5_auth_con_setlocalsubkey.3
krb5_auth_con_setrcache.3
krb5_auth_con_setremoteseqnumber.3
krb5_auth_con_setremotesubkey.3
krb5_auth_con_setuserkey.3
krb5_auth_context.3
krb5_auth_getremoteseqnumber.3
krb5_build_principal.3
krb5_build_principal_ext.3
krb5_build_principal_va.3
krb5_build_principal_va_ext.3
krb5_c_block_size.3
krb5_c_checksum_length.3
krb5_c_decrypt.3
krb5_c_encrypt.3
krb5_c_encrypt_length.3
krb5_c_enctype_compare.3
krb5_c_get_checksum.3
krb5_c_is_coll_proof_cksum.3
krb5_c_is_keyed_cksum.3
krb5_c_make_checksum.3
krb5_c_make_random_key.3
krb5_c_set_checksum.3
krb5_c_valid_cksumtype.3
krb5_c_valid_enctype.3
krb5_c_verify_checksum.3
krb5_cc_cache_end_seq_get.3
krb5_cc_cache_get_first.3
krb5_cc_cache_match.3
krb5_cc_cache_next.3
krb5_cc_close.3
krb5_cc_copy_cache.3
krb5_cc_default.3
krb5_cc_default_name.3
krb5_cc_destroy.3
krb5_cc_end_seq_get.3
krb5_cc_gen_new.3
krb5_cc_get_full_name.3

krb5_cc_get_name.3
krb5_cc_get_ops.3
krb5_cc_get_principal.3
krb5_cc_get_type.3
krb5_cc_get_version.3
krb5_cc_initialize.3
krb5_cc_new_unique.3
krb5_cc_next_cred.3
krb5_cc_register.3
krb5_cc_remove_cred.3
krb5_cc_resolve.3
krb5_cc_retrieve_cred.3
krb5_cc_set_default_name.3
krb5_cc_set_flags.3
krb5_cc_store_cred.3
krb5_change_password.3
krb5_check_transited.3
krb5_check_transited_realms.3
krb5_checksum_disable.3
krb5_checksum_free.3
krb5_checksum_is_collision_proof.3
krb5_checksum_is_keyed.3
krb5_checksumsize.3
krb5_clear_error_string.3
krb5_closelog.3
krb5_config_file_free.3
krb5_config_free_strings.3
krb5_config_get.3
krb5_config_get_bool.3
krb5_config_get_bool_default.3
krb5_config_get_int.3
krb5_config_get_int_default.3
krb5_config_get_list.3
krb5_config_get_next.3
krb5_config_get_string.3
krb5_config_get_string_default.3
krb5_config_get_strings.3
krb5_config_get_time.3
krb5_config_get_time_default.3
krb5_config_parse_file.3
krb5_config_parse_file_multi.3
krb5_config_vget.3
krb5_config_vget_bool.3
krb5_config_vget_bool_default.3
krb5_config_vget_int.3
krb5_config_vget_int_default.3
krb5_config_vget_list.3
krb5_config_vget_next.3
krb5_config_vget_string.3
krb5_config_vget_string_default.3
krb5_config_vget_strings.3
krb5_config_vget_time.3

krb5_config_vget_time_default.3
krb5_context.3
krb5_copy_address.3
krb5_copy_addresses.3
krb5_copy_checksum.3
krb5_copy_data.3
krb5_copy_host_realm.3
krb5_copy_keyblock.3
krb5_copy_keyblock_contents.3
krb5_copy_principal.3
krb5_copy_ticket.3
krb5_create_checksum.3
krb5_creds.3
krb5_crypto_destroy.3
krb5_crypto_get_checksum_type.3
krb5_crypto_getblocksize.3
krb5_crypto_getconfoundersize.3
krb5_crypto_getenctype.3
krb5_crypto_getpadsize.3
krb5_crypto_init.3
krb5_data_alloc.3
krb5_data_copy.3
krb5_data_free.3
krb5_data_realloc.3
krb5_data_zero.3
krb5_decrypt.3
krb5_decrypt_EncryptedData.3
krb5_digest.3
krb5_digest_alloc.3
krb5_digest_free.3
krb5_digest_get_a1_hash.3
krb5_digest_get_client_binding.3
krb5_digest_get_identifier.3
krb5_digest_get_opaque.3
krb5_digest_get_responseData.3
krb5_digest_get_rsp.3
krb5_digest_get_server_nonce.3
krb5_digest_get_tickets.3
krb5_digest_init_request.3
krb5_digest_request.3
krb5_digest_set_authentication_user.3
krb5_digest_set_authid.3
krb5_digest_set_client_nonce.3
krb5_digest_set_digest.3
krb5_digest_set_hostname.3
krb5_digest_set_identifier.3
krb5_digest_set_method.3
krb5_digest_set_nonceCount.3
krb5_digest_set_opaque.3
krb5_digest_set_qop.3
krb5_digest_set_realm.3
krb5_digest_set_server_cb.3

krb5_digest_set_server_nonce.3
krb5_digest_set_type.3
krb5_digest_set_uri.3
krb5_digest_set_username.3
krb5_domain_x500_decode.3
krb5_domain_x500_encode.3
krb5_eai_to_heim_errno.3
krb5_encrypt.3
krb5_encrypt_EncryptedData.3
krb5_enctype_disable.3
krb5_enctype_to_string.3
krb5_enctype_valid.3
krb5_err.3
krb5_errx.3
krb5_expand_hostname.3
krb5_expand_hostname_realms.3
krb5_find_padata.3
krb5_format_time.3
krb5_free_address.3
krb5_free_addresses.3
krb5_free_authenticator.3
krb5_free_checksum.3
krb5_free_checksum_contents.3
krb5_free_config_files.3
krb5_free_context.3
krb5_free_data.3
krb5_free_data_contents.3
krb5_free_error_string.3
krb5_free_host_realm.3
krb5_free_kdc_rep.3
krb5_free_keyblock.3
krb5_free_keyblock_contents.3
krb5_free_krbhst.3
krb5_free_principal.3
krb5_free_salt.3
krb5_free_ticket.3
krb5_fwd_tgt_creds.3
krb5_generate_random_block.3
krb5_generate_random_keyblock.3
krb5_generate_subkey.3
krb5_get_all_client_addrs.3
krb5_get_all_server_addrs.3
krb5_get_cred_from_kdc.3
krb5_get_cred_from_kdc_opt.3
krb5_get_credentials.3
krb5_get_credentials_with_flags.3
krb5_get_default_config_files.3
krb5_get_default_principal.3
krb5_get_default_realm.3
krb5_get_default_realms.3
krb5_get_err_text.3
krb5_get_error_message.3

krb5_get_error_string.3
krb5_get_extra_addresses.3
krb5_get_fcache_version.3
krb5_get_forwarded_creds.3
krb5_get_host_realm.3
krb5_get_ignore_addresses.3
krb5_get_in_cred.3
krb5_get_in_tkt.3
krb5_get_in_tkt_with_keytab.3
krb5_get_in_tkt_with_password.3
krb5_get_in_tkt_with_skey.3
krb5_get_init_creds.3
krb5_get_init_creds_keytab.3
krb5_get_init_creds_opt_alloc.3
krb5_get_init_creds_opt_free.3
krb5_get_init_creds_opt_free_pkinit.3
krb5_get_init_creds_opt_init.3
krb5_get_init_creds_opt_set_address_list.3
krb5_get_init_creds_opt_set_anonymous.3
krb5_get_init_creds_opt_set_default_flags.3
krb5_get_init_creds_opt_set_etype_list.3
krb5_get_init_creds_opt_set_forwardable.3
krb5_get_init_creds_opt_set_pa_password.3
krb5_get_init_creds_opt_set_paq_request.3
krb5_get_init_creds_opt_set_pkinit.3
krb5_get_init_creds_opt_set_preauth_list.3
krb5_get_init_creds_opt_set_proxiable.3
krb5_get_init_creds_opt_set_renew_life.3
krb5_get_init_creds_opt_set_salt.3
krb5_get_init_creds_opt_set_tkt_life.3
krb5_get_init_creds_password.3
krb5_get_kdc_cred.3
krb5_get_krb524hst.3
krb5_get_krb_admin_hst.3
krb5_get_krb_changepw_hst.3
krb5_get_krbhst.3
krb5_get_pw_salt.3
krb5_get_server_rcache.3
krb5_get_use_admin_kdc.3
krb5_get_wrapped_length.3
krb5_getportbyname.3
krb5_h_addr2addr.3
krb5_h_addr2sockaddr.3
krb5_h_errno_to_heim_errno.3
krb5_have_error_string.3
krb5_hmac.3
krb5_init_context.3
krb5_init_ets.3
krb5_initlog.3
krb5_keyblock_get_enctype.3
krb5_keyblock_zero.3
krb5_keytab_entry.3

krb5_krbhst_format_string.3
krb5_krbhst_free.3
krb5_krbhst_get_addrinfo.3
krb5_krbhst_init.3
krb5_krbhst_init_flags.3
krb5_krbhst_next.3
krb5_krbhst_next_as_string.3
krb5_krbhst_reset.3
krb5_kt_add_entry.3
krb5_kt_close.3
krb5_kt_compare.3
krb5_kt_copy_entry_contents.3
krb5_kt_cursor.3
krb5_kt_default.3
krb5_kt_default_modify_name.3
krb5_kt_default_name.3
krb5_kt_end_seq_get.3
krb5_kt_free_entry.3
krb5_kt_get_entry.3
krb5_kt_get_name.3
krb5_kt_get_type.3
krb5_kt_next_entry.3
krb5_kt_ops.3
krb5_kt_read_service_key.3
krb5_kt_register.3
krb5_kt_remove_entry.3
krb5_kt_resolve.3.3
krb5_kt_start_seq_get
krb5_kuserok.3
krb5_log.3
krb5_log_msg.3
krb5_make_addrport.3
krb5_make_principal.3
krb5_max_sockaddr_size.3
krb5_openlog.3
krb5_padata_add.3
krb5_parse_address.3
krb5_parse_name.3
krb5_passwd_result_to_string.3
krb5_password_key_proc.3
krb5_prepend_config_files.3
krb5_prepend_config_files_default.3
krb5_princ_realm.3
krb5_princ_set_realm.3
krb5_principal.3
krb5_principal_compare.3
krb5_principal_compare_any_realm.3
krb5_principal_get_comp_string.3
krb5_principal_get_realm.3
krb5_principal_get_type.3
krb5_principal_match.3
krb5_principal_set_type.3

krb5_print_address.3
krb5_rc_close.3
krb5_rc_default.3
krb5_rc_default_name.3
krb5_rc_default_type.3
krb5_rc_destroy.3
krb5_rc_expunge.3
krb5_rc_get_lifespan.3
krb5_rc_get_name.3
krb5_rc_get_type.3
krb5_rc_initialize.3
krb5_rc_recover.3
krb5_rc_resolve.3
krb5_rc_resolve_full.3
krb5_rc_resolve_type.3
krb5_rc_store.3
krb5_rcache.3
krb5_realm_compare.3
krb5_ret_address.3
krb5_ret_addrs.3
krb5_ret_authdata.3
krb5_ret_creds.3
krb5_ret_data.3
krb5_ret_int16.3
krb5_ret_int32.3
krb5_ret_int8.3
krb5_ret_keyblock.3
krb5_ret_principal.3
krb5_ret_string.3
krb5_ret_stringz.3
krb5_ret_times.3
krb5_set_config_files.3
krb5_set_default_realm.3
krb5_set_error_string.3
krb5_set_extra_addresses.3
krb5_set_fcache_version.3
krb5_set_ignore_addresses.3
krb5_set_password.3
krb5_set_password_using_ccache.3
krb5_set_real_time.3
krb5_set_use_admin_kdc.3
krb5_set_warn_dest.3
krb5_sname_to_principal.3
krb5_sock_to_principal.3
krb5_sockaddr2address.3
krb5_sockaddr2port.3
krb5_sockaddr_uninteresting.3
krb5_storage.3
krb5_storage_clear_flags.3
krb5_storage_emem.3
krb5_storage_free.3
krb5_storage_from_data.3

krb5_storage_from_fd.3
krb5_storage_from_mem.3
krb5_storage_get_byteorder.3
krb5_storage_is_flags.3
krb5_storage_read.3
krb5_storage_seek.3
krb5_storage_set_byteorder.3
krb5_storage_set_eof_code.3
krb5_storage_set_flags.3
krb5_storage_to_data.3
krb5_storage_write.3
krb5_store_address.3
krb5_store_addrs.3
krb5_store_authdata.3
krb5_store_creds.3
krb5_store_data.3
krb5_store_int16.3
krb5_store_int32.3
krb5_store_int8.3
krb5_store_keyblock.3
krb5_store_principal.3
krb5_store_string.3
krb5_store_stringz.3
krb5_store_times.3
krb5_string_to_deltat.3
krb5_string_to_enctype.3
krb5_string_to_key.3
krb5_string_to_key_data.3
krb5_string_to_key_data_salt.3
krb5_string_to_key_data_salt_opaque.3
krb5_string_to_key_salt.3
krb5_string_to_key_salt_opaque.3
krb5_ticket.3
krb5_ticket_get_authorization_data_type.3
krb5_ticket_get_client.3
krb5_ticket_get_server.3
krb5_timeofday.3
krb5_unparse_name.3
krb5_unparse_name_fixed.3
krb5_unparse_name_fixed_short.3
krb5_unparse_name_short.3
krb5_us_timeofday.3
krb5_vabort.3
krb5_vabortx.3
krb5_verify_checksum.3
krb5_verify_init_creds.3
krb5_verify_init_creds_opt_init.3
krb5_verify_init_creds_opt_set_ap_req_nofail.3
krb5_verify_opt_init.3
krb5_verify_opt_set_ccache.3
krb5_verify_opt_set_flags.3
krb5_verify_opt_set_keytab.3

krb5_verify_opt_set_secure.3
krb5_verify_opt_set_service.3
krb5_verify_user.3
krb5_verify_user_lrealm.3
krb5_verify_user_opt.3
krb5_verr.3
krb5_verrx.3
krb5_vlog.3
krb5_vlog_msg.3
krb5_vset_error_string.3
krb5_vwarn.3
krb5_vwarnx.3
krb5_warn.3
krb5_warnx.3

SEE ALSO

`krb5.conf`(5), `kerberos`(8)

NAME

krb524_convert_creds_kdc, **krb524_convert_creds_kdc_ccache** — converts Kerberos 5 credentials to Kerberos 4 credentials

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb524_convert_creds_kdc(krb5_context context, krb5_creds *in_cred,
    struct credentials *v4creds);

krb5_error_code
krb524_convert_creds_kdc_ccache(krb5_context context, krb5_ccache ccache,
    krb5_creds *in_cred, struct credentials *v4creds);
```

DESCRIPTION

Convert the Kerberos 5 credential to Kerberos 4 credential. This is done by sending them to the 524 service in the KDC.

krb524_convert_creds_kdc() converts the Kerberos 5 credential in *in_cred* to Kerberos 4 credential that is stored in *credentials*.

krb524_convert_creds_kdc_ccache() is diffrent from **krb524_convert_creds_kdc**() in that way that if *in_cred* doesn't contain a DES session key, then a new one is fetched from the KDC and stored in the cred cache *ccache*, and then the KDC is queried to convert the credential.

This interfaces are used to make the migration to Kerberos 5 from Kerberos 4 easier. There are few services that still need Kerberos 4, and this is mainly for compatibility for those services. Some services, like AFS, really have Kerberos 5 supports, but still uses the 524 interface to make the migration easier.

SEE ALSO

krb5(3), krb5.conf(5)

NAME

krb5_425_conv_principal, **krb5_425_conv_principal_ext**, **krb5_524_conv_principal** — converts to and from version 4 principals

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_425_conv_principal(krb5_context context, const char *name,
    const char *instance, const char *realm, krb5_principal *principal);

krb5_error_code
krb5_425_conv_principal_ext(krb5_context context, const char *name,
    const char *instance, const char *realm,
    krb5_boolean (*func)(krb5_context, krb5_principal),
    krb5_boolean resolve, krb5_principal *principal);

krb5_error_code
krb5_524_conv_principal(krb5_context context,
    const krb5_principal principal, char *name, char *instance,
    char *realm);
```

DESCRIPTION

Converting between version 4 and version 5 principals can at best be described as a mess.

A version 4 principal consists of a name, an instance, and a realm. A version 5 principal consists of one or more components, and a realm. In some cases also the first component/name will differ between version 4 and version 5. Furthermore the second component of a host principal will be the fully qualified domain name of the host in question, while the instance of a version 4 principal will only contain the first part (short hostname). Because of these problems the conversion between principals will have to be site customized.

krb5_425_conv_principal_ext() will try to convert a version 4 principal, given by *name*, *instance*, and *realm*, to a version 5 principal. This can result in several possible principals, and if *func* is non-NULL, it will be called for each candidate principal. *func* should return true if the principal was "good". To accomplish this, **krb5_425_conv_principal_ext**() will look up the name in `krb5.conf`. It first looks in the `v4_name_convert/host` subsection, which should contain a list of version 4 names whose instance should be treated as a hostname. This list can be specified for each realm (in the `realms` section), or in the `libdefaults` section. If the name is found the resulting name of the principal will be the value of this binding. The instance is then first looked up in `v4_instance_convert` for the specified realm. If found the resulting value will be used as instance (this can be used for special cases), no further attempts will be made to find a conversion if this fails (with *func*). If the *resolve* parameter is true, the instance will be looked up with **gethostbyname**(). This can be a time consuming, error prone, and unsafe operation. Next a list of hostnames will be created from the instance and the `v4_domains` variable, which should contain a list of possible domains for the specific realm.

On the other hand, if the name is not found in a `host` section, it is looked up in a `v4_name_convert/plain` binding. If found here the name will be converted, but the instance will be untouched.

This list of default host-type conversions is compiled-in:

```
v4_name_convert = {
        host = {
                ftp = ftp
                hprop = hprop
                imap = imap
                pop = pop
                rcmd = host
                smtp = smtp
        }
}
```

It will only be used if there isn't an entry for these names in the config file, so you can override these defaults.

krb5_425_conv_principal() will call **krb5_425_conv_principal_ext**() with NULL as *func*, and the value of v4_instance_resolve (from the libdefaults section) as *resolve*.

krb5_524_conv_principal() basically does the opposite of **krb5_425_conv_principal**(), it just doesn't have to look up any names, but will instead truncate instances found to belong to a host principal. The *name*, *instance*, and *realm* should be at least 40 characters long.

EXAMPLES

Since this is confusing an example is in place.

Assume that we have the "foo.com", and "bar.com" domains that have shared a single version 4 realm, FOO.COM. The version 4 krb.realms file looked like:

```
foo.com         FOO.COM
.foo.com        FOO.COM
.bar.com        FOO.COM
```

A krb5.conf file that covers this case might look like:

```
[libdefaults]
        v4_instance_resolve = yes
[realms]
        FOO.COM = {
                kdc = kerberos.foo.com
                v4_instance_convert = {
                        foo = foo.com
                }
                v4_domains = foo.com
        }
```

With this setup and the following host table:

```
foo.com
a-host.foo.com
b-host.bar.com
```

the following conversions will be made:

```
rcmd.a-host     -> host/a-host.foo.com
ftp.b-host      -> ftp/b-host.bar.com
pop.foo         -> pop/foo.com
ftp.other       -> ftp/other.foo.com
other.a-host    -> other/a-host
```

The first three are what you expect. If you remove the “v4_domains”, the fourth entry will result in an error (since the host “other” can’t be found). Even if “a-host” is a valid host name, the last entry will not be converted, since the “other” name is not known to represent a host-type principal. If you turn off “v4_instance_resolve” the second example will result in “ftp/b-host.foo.com” (because of the default domain). And all of this is of course only valid if you have working name resolving.

SEE ALSO

`krb5_build_principal`(3), `krb5_free_principal`(3), `krb5_parse_name`(3), `krb5_sname_to_principal`(3), `krb5_unparse_name`(3), `krb5.conf`(5)

NAME

krb5_acl_match_file, **krb5_acl_match_string** — ACL matching functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
krb5_error_code
krb5_acl_match_file(krb5_context context, const char *file,
    const char *format, ...);

krb5_error_code
krb5_acl_match_string(krb5_context context, const char *string,
    const char *format, ...);
```

DESCRIPTION

krb5_acl_match_file matches ACL format against each line in a file. Lines starting with # are treated like comments and ignored.

krb5_acl_match_string matches ACL format against a string.

The ACL format has three format specifiers: s, f, and r. Each specifier will retrieve one argument from the variable arguments for either matching or storing data. The input string is split up using " " and "\t" as a delimiter; multiple " " and "\t" in a row are considered to be the same.

- s Matches a string using `strcmp`(3) (case sensitive).
- f Matches the string with `fnmatch`(3). The *flags* argument (the last argument) passed to the fnmatch function is 0.
- r Returns a copy of the string in the char ** passed in; the copy must be freed with `free`(3). There is no need to `free`(3) the string on error: the function will clean up and set the pointer to `NULL`.

All unknown format specifiers cause an error.

EXAMPLES

```
char *s;

ret = krb5_acl_match_string(context, "foo", "s", "foo");
if (ret)
    krb5_errx(context, 1, "acl didn't match");
ret = krb5_acl_match_string(context, "foo foo baz/kaka",
    "ss", "foo", &s, "foo/*");
if (ret) {
    /* no need to free(s) on error */
    assert(s == NULL);
    krb5_errx(context, 1, "acl didn't match");
}
free(s);
```

SEE ALSO

`krb5`(3)

NAME

krb5_address, **krb5_addresses**, **krb5_sockaddr2address**, **krb5_sockaddr2port**, **krb5_addr2sockaddr**, **krb5_max_sockaddr_size**, **krb5_sockaddr_uninteresting**, **krb5_h_addr2sockaddr**, **krb5_h_addr2addr**, **krb5_anyaddr**, **krb5_print_address**, **krb5_parse_address**, **krb5_address_order**, **krb5_address_compare**, **krb5_address_search**, **krb5_free_address**, **krb5_free_addresses**, **krb5_copy_address**, **krb5_copy_addresses**, **krb5_append_addresses**, **krb5_make_addrport** — mange addresses in Kerberos

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_sockaddr2address(krb5_context context, const struct sockaddr *sa,
    krb5_address *addr);

krb5_error_code
krb5_sockaddr2port(krb5_context context, const struct sockaddr *sa,
    int16_t *port);

krb5_error_code
krb5_addr2sockaddr(krb5_context context, const krb5_address *addr,
    struct sockaddr *sa, krb5_socklen_t *sa_size, int port);

size_t
krb5_max_sockaddr_size(void);

krb5_boolean
krb5_sockaddr_uninteresting(const struct sockaddr *sa);

krb5_error_code
krb5_h_addr2sockaddr(krb5_context context, int af, const char *addr,
    struct sockaddr *sa, krb5_socklen_t *sa_size, int port);

krb5_error_code
krb5_h_addr2addr(krb5_context context, int af, const char *haddr,
    krb5_address *addr);

krb5_error_code
krb5_anyaddr(krb5_context context, int af, struct sockaddr *sa,
    krb5_socklen_t *sa_size, int port);

krb5_error_code
krb5_print_address(const krb5_address *addr, char *str, size_t len,
    size_t *ret_len);

krb5_error_code
krb5_parse_address(krb5_context context, const char *string,
    krb5_addresses *addresses);

int
krb5_address_order(krb5_context context, const krb5_address *addr1,
    const krb5_address *addr2);
```

krb5_boolean
krb5_address_compare(*krb5_context context*, *const krb5_address *addr1*, *const krb5_address *addr2*);

krb5_boolean
krb5_address_search(*krb5_context context*, *const krb5_address *addr*, *const krb5_addresses *addrlist*);

krb5_error_code
krb5_free_address(*krb5_context context*, *krb5_address *address*);

krb5_error_code
krb5_free_addresses(*krb5_context context*, *krb5_addresses *addresses*);

krb5_error_code
krb5_copy_address(*krb5_context context*, *const krb5_address *inaddr*, *krb5_address *outaddr*);

krb5_error_code
krb5_copy_addresses(*krb5_context context*, *const krb5_addresses *inaddr*, *krb5_addresses *outaddr*);

krb5_error_code
krb5_append_addresses(*krb5_context context*, *krb5_addresses *dest*, *const krb5_addresses *source*);

krb5_error_code
krb5_make_addrport(*krb5_context context*, *krb5_address **res*, *const krb5_address *addr*, *int16_t port*);

DESCRIPTION

The krb5_address structure holds a address that can be used in Kerberos API calls. There are help functions to set and extract address information of the address.

The krb5_addresses structure holds a set of krb5_address:es.

krb5_sockaddr2address() stores a address a struct sockaddr *sa* in the krb5_address *addr*.

krb5_sockaddr2port() extracts a *port* (if possible) from a struct sockaddr *sa*.

krb5_addr2sockaddr() sets the struct sockaddr *sockaddr* from *addr* and *port*. The argument *sa_size* should initially contain the size of the *sa*, and after the call, it will contain the actual length of the address.

krb5_max_sockaddr_size() returns the max size of the struct sockaddr that the Kerberos library will return.

krb5_sockaddr_uninteresting() returns TRUE for all *sa* that the kerberos library thinks are uninteresting. One example are link local addresses.

krb5_h_addr2sockaddr() initializes a struct sockaddr *sa* from *af* and the struct hostent (see gethostbyname(3)) *h_addr_list* component. The argument *sa_size* should initially contain the size of the *sa*, and after the call, it will contain the actual length of the address.

krb5_h_addr2addr() works like **krb5_h_addr2sockaddr**() with the exception that it operates on a krb5_address instead of a struct sockaddr.

krb5_anyaddr() fills in a struct sockaddr *sa* that can be used to bind(2) to. The argument *sa_size* should initially contain the size of the *sa*, and after the call, it will contain the actual length of the address.

krb5_print_address() prints the address in *addr* to the string *string* that have the length *len*. If *ret_len* is not NULL, it will be filled with the length of the string if size were unlimited (not including the final '\0').

krb5_parse_address() Returns the resolved hostname in *string* to the krb5_addresses *addresses*.

krb5_address_order() compares the addresses *addr1* and *addr2* so that it can be used for sorting addresses. If the addresses are the same address *krb5_address_order* will return 0.

krb5_address_compare() compares the addresses *addr1* and *addr2*. Returns TRUE if the two addresses are the same.

krb5_address_search() checks if the address *addr* is a member of the address set list *addrlist*.

krb5_free_address() frees the data stored in the *address* that is alloced with any of the krb5_address functions.

krb5_free_addresses() frees the data stored in the *addresses* that is alloced with any of the krb5_address functions.

krb5_copy_address() copies the content of address *inaddr* to *outaddr*.

krb5_copy_addresses() copies the content of the address list *inaddr* to *outaddr*.

krb5_append_addresses() adds the set of addresses in *source* to *dest*. While copying the addresses, duplicates are also sorted out.

krb5_make_addrport() allocates and creates an krb5_address in *res* of type KRB5_ADDRESS_ADDRPORT from (*addr*, *port*).

SEE ALSO

krb5(3), krb5.conf(5), kerberos(8)

NAME

krb5_aname_to_localname — converts a principal to a system local name

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_boolean
krb5_aname_to_localname(krb5_context context, krb5_const_principal name,
    size_t lnsize, char *lname);
```

DESCRIPTION

This function takes a principal *name*, verifies that it is in the local realm (using **krb5_get_default_realms**()) and then returns the local name of the principal.

If *name* isn't in one of the local realms an error is returned.

If the size (*lnsize*) of the local name (*lname*) is too small, an error is returned.

krb5_aname_to_localname() should only be use by an application that implements protocols that don't transport the login name and thus needs to convert a principal to a local name.

Protocols should be designed so that they authenticate using Kerberos, send over the login name and then verify the principal that is authenticated is allowed to login and the login name. A way to check if a user is allowed to login is using the function **krb5_kuserok**().

SEE ALSO

krb5_get_default_realms(3), krb5_kuserok(3)

NAME

krb5_appdefault_boolean, **krb5_appdefault_string**, **krb5_appdefault_time** — get application configuration value

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

void
krb5_appdefault_boolean(krb5_context context, const char *appname,
    krb5_realm realm, const char *option, krb5_boolean def_val,
    krb5_boolean *ret_val);

void
krb5_appdefault_string(krb5_context context, const char *appname,
    krb5_realm realm, const char *option, const char *def_val,
    char **ret_val);

void
krb5_appdefault_time(krb5_context context, const char *appname,
    krb5_realm realm, const char *option, time_t def_val, time_t *ret_val);
```

DESCRIPTION

These functions get application defaults from the `appdefaults` section of the `krb5.conf`(5) configuration file. These defaults can be specified per application, and/or per realm.

These values will be looked for in `krb5.conf`(5), in order of descending importance.

```
[appdefaults]
        appname = {
                realm = {
                        option = value
                }
        }
        appname = {
                option = value
        }
        realm = {
                option = value
        }
        option = value
```

appname is the name of the application, and *realm* is the realm name. If the realm is omitted it will not be used for resolving values. *def_val* is the value to return if no value is found in `krb5.conf`(5).

SEE ALSO

`krb5_config`(3), `krb5.conf`(5)

NAME

krb5_auth_con_addflags, **krb5_auth_con_free**, **krb5_auth_con_genaddrs**, **krb5_auth_con_generatelocalsubkey**, **krb5_auth_con_getaddrs**, **krb5_auth_con_getauthenticator**, **krb5_auth_con_getflags**, **krb5_auth_con_getkey**, **krb5_auth_con_getlocalsubkey**, **krb5_auth_con_getrcache**, **krb5_auth_con_getremotesubkey**, **krb5_auth_con_getuserkey**, **krb5_auth_con_init**, **krb5_auth_con_initivector**, **krb5_auth_con_removeflags**, **krb5_auth_con_setaddrs**, **krb5_auth_con_setaddrs_from_fd**, **krb5_auth_con_setflags**, **krb5_auth_con_setivector**, **krb5_auth_con_setkey**, **krb5_auth_con_setlocalsubkey**, **krb5_auth_con_setrcache**, **krb5_auth_con_setremotesubkey**, **krb5_auth_con_setuserkey**, **krb5_auth_context**, **krb5_auth_getcksumtype**, **krb5_auth_getkeytype**, **krb5_auth_getlocalseqnumber**, **krb5_auth_getremoteseqnumber**, **krb5_auth_setcksumtype**, **krb5_auth_setkeytype**, **krb5_auth_setlocalseqnumber**, **krb5_auth_setremoteseqnumber**, **krb5_free_authenticator** — manage authentication on connection level

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_auth_con_init(krb5_context context, krb5_auth_context *auth_context);

void
krb5_auth_con_free(krb5_context context, krb5_auth_context auth_context);

krb5_error_code
krb5_auth_con_setflags(krb5_context context,
    krb5_auth_context auth_context, int32_t flags);

krb5_error_code
krb5_auth_con_getflags(krb5_context context,
    krb5_auth_context auth_context, int32_t *flags);

krb5_error_code
krb5_auth_con_addflags(krb5_context context,
    krb5_auth_context auth_context, int32_t addflags, int32_t *flags);

krb5_error_code
krb5_auth_con_removeflags(krb5_context context,
    krb5_auth_context auth_context, int32_t removelags, int32_t *flags);

krb5_error_code
krb5_auth_con_setaddrs(krb5_context context,
    krb5_auth_context auth_context, krb5_address *local_addr,
    krb5_address *remote_addr);

krb5_error_code
krb5_auth_con_getaddrs(krb5_context context,
    krb5_auth_context auth_context, krb5_address **local_addr,
    krb5_address **remote_addr);
```

krb5_error_code
krb5_auth_con_genaddrs(*krb5_context context*,
krb5_auth_context auth_context,*int fd*,*int flags*);

krb5_error_code
krb5_auth_con_setaddrs_from_fd(*krb5_context context*,
krb5_auth_context auth_context,*void *p_fd*);

krb5_error_code
krb5_auth_con_getkey(*krb5_context context*,*krb5_auth_context auth_context*,
*krb5_keyblock **keyblock*);

krb5_error_code
krb5_auth_con_getlocalsubkey(*krb5_context context*,
krb5_auth_context auth_context,*krb5_keyblock **keyblock*);

krb5_error_code
krb5_auth_con_getremotesubkey(*krb5_context context*,
krb5_auth_context auth_context,*krb5_keyblock **keyblock*);

krb5_error_code
krb5_auth_con_generatelocalsubkey(*krb5_context context*,
krb5_auth_context auth_context,*krb5_keyblock*,*key"*);

krb5_error_code
krb5_auth_con_initivector(*krb5_context context*,
krb5_auth_context auth_context);

krb5_error_code
krb5_auth_con_setivector(*krb5_context context*,
*krb5_auth_context *auth_context*,*krb5_pointer ivector*);

void
krb5_free_authenticator(*krb5_context context*,
*krb5_authenticator *authenticator*);

DESCRIPTION

The **krb5_auth_context** structure holds all context related to an authenticated connection, in a similar way to **krb5_context** that holds the context for the thread or process. **krb5_auth_context** is used by various functions that are directly related to authentication between the server/client. Example of data that this structure contains are various flags, addresses of client and server, port numbers, keyblocks (and subkeys), sequence numbers, replay cache, and checksum-type.

krb5_auth_con_init() allocates and initializes the **krb5_auth_context** structure. Default values can be changed with **krb5_auth_con_setcksumtype**() and **krb5_auth_con_setflags**(). The **auth_context** structure must be freed by **krb5_auth_con_free**().

krb5_auth_con_getflags(), **krb5_auth_con_setflags**(), **krb5_auth_con_addflags**() and **krb5_auth_con_removeflags**() gets and modifies the flags for a **krb5_auth_context** structure. Possible flags to set are:

KRB5_AUTH_CONTEXT_DO_SEQUENCE
: Generate and check sequence-number on each packet.

KRB5_AUTH_CONTEXT_DO_TIME
: Check timestamp on incoming packets.

KRB5_AUTH_CONTEXT_RET_SEQUENCE, KRB5_AUTH_CONTEXT_RET_TIME
Return sequence numbers and time stamps in the outdata parameters.

KRB5_AUTH_CONTEXT_CLEAR_FORWARDED_CRED
will force **krb5_get_forwarded_creds**() and **krb5_fwd_tgt_creds**() to create unencrypted) ENCTYPE_NULL) credentials. This is for use with old MIT server and JAVA based servers as they can't handle encrypted KRB-CRED. Note that sending such KRB-CRED is clear exposes crypto keys and tickets and is insecure, make sure the packet is encrypted in the protocol. krb5_rd_cred(3), krb5_rd_priv(3), krb5_rd_safe(3), krb5_mk_priv(3) and krb5_mk_safe(3). Setting this flag requires that parameter to be passed to these functions.

The flags KRB5_AUTH_CONTEXT_DO_TIME also modifies the behavior the function **krb5_get_forwarded_creds**() by removing the timestamp in the forward credential message, this have backward compatibility problems since not all versions of the heimdal supports timeless credentional messages. Is very useful since it always the sender of the message to cache forward message and thus avoiding a round trip to the KDC for each time a credential is forwarded. The same functionality can be obtained by using address-less tickets.

krb5_auth_con_setaddrs(), **krb5_auth_con_setaddrs_from_fd**() and **krb5_auth_con_getaddrs**() gets and sets the addresses that are checked when a packet is received. It is mandatory to set an address for the remote host. If the local address is not set, it iss deduced from the underlaying operating system. **krb5_auth_con_getaddrs**() will call **krb5_free_address**() on any address that is passed in *local_addr* or *remote_addr*. **krb5_auth_con_setaddr**() allows passing in a NULL pointer as *local_addr* and *remote_addr*, in that case it will just not set that address.

krb5_auth_con_setaddrs_from_fd() fetches the addresses from a file descriptor.

krb5_auth_con_genaddrs() fetches the address information from the given file descriptor *fd* depending on the bitmap argument *flags*.

Possible values on *flags* are:

KRB5_AUTH_CONTEXT_GENERATE_LOCAL_ADDR
fetches the local address from *fd*.

KRB5_AUTH_CONTEXT_GENERATE_REMOTE_ADDR
fetches the remote address from *fd*.

krb5_auth_con_setkey(), **krb5_auth_con_setuserkey**() and **krb5_auth_con_getkey**() gets and sets the key used for this auth context. The keyblock returned by **krb5_auth_con_getkey**() should be freed with **krb5_free_keyblock**(). The keyblock send into **krb5_auth_con_setkey**() is copied into the **krb5_auth_context**, and thus no special handling is needed. NULL is not a valid keyblock to **krb5_auth_con_setkey**().

krb5_auth_con_setuserkey() is only useful when doing user to user authentication. **krb5_auth_con_setkey**() is equivalent to **krb5_auth_con_setuserkey**().

krb5_auth_con_getlocalsubkey(), **krb5_auth_con_setlocalsubkey**(), **krb5_auth_con_getremotesubkey**() and **krb5_auth_con_setremotesubkey**() gets and sets the keyblock for the local and remote subkey. The keyblock returned by **krb5_auth_con_getlocalsubkey**() and **krb5_auth_con_getremotesubkey**() must be freed with **krb5_free_keyblock**().

krb5_auth_setcksumtype() and **krb5_auth_getcksumtype**() sets and gets the checksum type that should be used for this connection.

krb5_auth_con_generatelocalsubkey() generates a local subkey that have the same encryption type as *key*.

krb5_auth_getremoteseqnumber() **krb5_auth_setremoteseqnumber**(), **krb5_auth_getlocalseqnumber**() and **krb5_auth_setlocalseqnumber**() gets and sets the sequence-number for the local and remote sequence-number counter.

krb5_auth_setkeytype() and **krb5_auth_getkeytype**() gets and gets the keytype of the keyblock in **krb5_auth_context**.

krb5_auth_con_getauthenticator() Retrieves the authenticator that was used during mutual authentication. The `authenticator` returned should be freed by calling **krb5_free_authenticator**().

krb5_auth_con_getrcache() and **krb5_auth_con_setrcache**() gets and sets the replay-cache.

krb5_auth_con_initivector() allocates memory for and zeros the initial vector in the *`auth_context`* keyblock.

krb5_auth_con_setivector() sets the i_vector portion of *`auth_context`* to *`ivector`*.

krb5_free_authenticator() free the content of *`authenticator`* and *`authenticator`* itself.

SEE ALSO

`krb5_context`(3), `kerberos`(8)

NAME

krb5_c_block_size, **krb5_c_decrypt**, **krb5_c_encrypt**, **krb5_c_encrypt_length**, **krb5_c_enctype_compare**, **krb5_c_get_checksum**, **krb5_c_is_coll_proof_cksum**, **krb5_c_is_keyed_cksum**, **krb5_c_keylength**, **krb5_c_make_checksum**, **krb5_c_make_random_key**, **krb5_c_set_checksum**, **krb5_c_valid_cksumtype**, **krb5_c_valid_enctype**, **krb5_c_verify_checksum**, **krb5_c_checksum_length** — Kerberos 5 crypto API

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_c_block_size(krb5_context context, krb5_enctype enctype,
    size_t *blocksize);

krb5_error_code
krb5_c_decrypt(krb5_context context, const krb5_keyblock key,
    krb5_keyusage usage, const krb5_data *ivec, krb5_enc_data *input,
    krb5_data *output);

krb5_error_code
krb5_c_encrypt(krb5_context context, const krb5_keyblock *key,
    krb5_keyusage usage, const krb5_data *ivec, const krb5_data *input,
    krb5_enc_data *output);

krb5_error_code
krb5_c_encrypt_length(krb5_context context, krb5_enctype enctype,
    size_t inputlen, size_t *length);

krb5_error_code
krb5_c_enctype_compare(krb5_context context, krb5_enctype e1,
    krb5_enctype e2, krb5_boolean *similar);

krb5_error_code
krb5_c_make_random_key(krb5_context context, krb5_enctype enctype,
    krb5_keyblock *random_key);

krb5_error_code
krb5_c_make_checksum(krb5_context context, krb5_cksumtype cksumtype,
    const krb5_keyblock *key, krb5_keyusage usage, const krb5_data *input,
    krb5_checksum *cksum);

krb5_error_code
krb5_c_verify_checksum(krb5_context context, const krb5_keyblock *key,
    krb5_keyusage usage, const krb5_data *data, const krb5_checksum *cksum,
    krb5_boolean *valid);

krb5_error_code
krb5_c_checksum_length(krb5_context context, krb5_cksumtype cksumtype,
    size_t *length);

krb5_error_code
krb5_c_get_checksum(krb5_context context, const krb5_checksum *cksum,
    krb5_cksumtype *type, krb5_data **data);
```

krb5_error_code
krb5_c_set_checksum(*krb5_context context*, *krb5_checksum *cksum*, *krb5_cksumtype type*, *const krb5_data *data*);

krb5_boolean
krb5_c_valid_enctype(*krb5_enctype*, *etype"*);

krb5_boolean
krb5_c_valid_cksumtype(*krb5_cksumtype ctype*);

krb5_boolean
krb5_c_is_coll_proof_cksum(*krb5_cksumtype ctype*);

krb5_boolean
krb5_c_is_keyed_cksum(*krb5_cksumtype ctype*);

krb5_error_code
krb5_c_keylengths(*krb5_context context*, *krb5_enctype enctype*, *size_t *inlength*, *size_t *keylength*);

DESCRIPTION

The functions starting with krb5_c are compat functions with MIT kerberos.

The krb5_enc_data structure holds and encrypted data. There are two public accessable members of krb5_enc_data. enctype that holds the encryption type of the data encrypted and ciphertext that is a *krb5_data* that might contain the encrypted data.

krb5_c_block_size() returns the blocksize of the encryption type.

krb5_c_decrypt() decrypts *input* and store the data in *output.* If *ivec* is NULL the default initialization vector for that encryption type will be used.

krb5_c_encrypt() encrypts the plaintext in *input* and store the ciphertext in *output*.

krb5_c_encrypt_length() returns the length the encrypted data given the plaintext length.

krb5_c_enctype_compare() compares to encryption types and returns if they use compatible encryption key types.

krb5_c_make_checksum() creates a checksum *cksum* with the checksum type *cksumtype* of the data in *data*. *key* and *usage* are used if the checksum is a keyed checksum type. Returns 0 or an error code.

krb5_c_verify_checksum() verifies the checksum of *data* in *cksum* that was created with *key* using the key usage *usage*. *verify* is set to non-zero if the checksum verifies correctly and zero if not. Returns 0 or an error code.

krb5_c_checksum_length() returns the length of the checksum.

krb5_c_set_checksum() sets the krb5_checksum structure given *type* and *data*. The content of *cksum* should be freeed with **krb5_c_free_checksum_contents**().

krb5_c_get_checksum() retrieves the components of the krb5_checksum. structure. *data* should be free with **krb5_free_data**(). If some either of *data* or *checksum* is not needed for the application, NULL can be passed in.

krb5_c_valid_enctype() returns true if *etype* is a valid encryption type.

krb5_c_valid_cksumtype() returns true if *ctype* is a valid checksum type.

krb5_c_is_keyed_cksum() return true if *ctype* is a keyed checksum type.

krb5_c_is_coll_proof_cksum() returns true if *ctype* is a collition proof checksum type.

krb5_c_keylengths() return the minimum length (*inlength*) bytes needed to create a key and the length (*keylength*) of the resulting key for the *enctype*.

SEE ALSO

krb5(3), krb5_create_checksum(3), krb5_free_data(3), kerberos(8)

NAME

krb5_ccache, **krb5_cc_cursor**, **krb5_cc_ops**, **krb5_fcc_ops**, **krb5_mcc_ops**, **krb5_cc_clear_mcred**, **krb5_cc_close**, **krb5_cc_copy_cache**, **krb5_cc_default**, **krb5_cc_default_name**, **krb5_cc_destroy**, **krb5_cc_end_seq_get**, **krb5_cc_gen_new**, **krb5_cc_get_full_name**, **krb5_cc_get_name**, **krb5_cc_get_ops**, **krb5_cc_get_prefix_ops**, **krb5_cc_get_principal**, **krb5_cc_get_type**, **krb5_cc_get_version**, **krb5_cc_initialize**, **krb5_cc_next_cred**, **krb5_cc_next_cred_match**, **krb5_cc_new_unique**, **krb5_cc_register**, **krb5_cc_remove_cred**, **krb5_cc_resolve**, **krb5_cc_retrieve_cred**, **krb5_cc_set_default_name**, **krb5_cc_set_flags**, **krb5_cc_start_seq_get**, **krb5_cc_store_cred** — mange credential cache

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

struct krb5_ccache;

struct krb5_cc_cursor;

struct krb5_cc_ops;

struct krb5_cc_ops *krb5_fcc_ops;

struct krb5_cc_ops *krb5_mcc_ops;

void
krb5_cc_clear_mcred(krb5_creds *mcred);

krb5_error_code
krb5_cc_close(krb5_context context, krb5_ccache id);

krb5_error_code
krb5_cc_copy_cache(krb5_context context, const krb5_ccache from,
    krb5_ccache to);

krb5_error_code
krb5_cc_default(krb5_context context, krb5_ccache *id);

const char *
krb5_cc_default_name(krb5_context context);

krb5_error_code
krb5_cc_destroy(krb5_context context, krb5_ccache id);

krb5_error_code
krb5_cc_end_seq_get(krb5_context context, const krb5_ccache id,
    krb5_cc_cursor *cursor);

krb5_error_code
krb5_cc_gen_new(krb5_context context, const krb5_cc_ops *ops,
    krb5_ccache *id);

krb5_error_code
krb5_cc_get_full_name(krb5_context context, krb5_ccache id, char **str);
```

*const char **
krb5_cc_get_name(*krb5_context context*, *krb5_ccache id*);

krb5_error_code
krb5_cc_get_principal(*krb5_context context*, *krb5_ccache id*,
*krb5_principal *principal*);

*const char **
krb5_cc_get_type(*krb5_context context*, *krb5_ccache id*);

*const krb5_cc_ops **
krb5_cc_get_ops(*krb5_context context*, *krb5_ccache id*);

*const krb5_cc_ops **
krb5_cc_get_prefix_ops(*krb5_context context*, *const char *prefix*);

krb5_error_code
krb5_cc_get_version(*krb5_context context*, *const krb5_ccache id*);

krb5_error_code
krb5_cc_initialize(*krb5_context context*, *krb5_ccache id*,
krb5_principal primary_principal);

krb5_error_code
krb5_cc_register(*krb5_context context*, *const krb5_cc_ops *ops*,
krb5_boolean override);

krb5_error_code
krb5_cc_resolve(*krb5_context context*, *const char *name*, *krb5_ccache *id*);

krb5_error_code
krb5_cc_retrieve_cred(*krb5_context context*, *krb5_ccache id*,
krb5_flags whichfields, *const krb5_creds *mcreds*, *krb5_creds *creds*);

krb5_error_code
krb5_cc_remove_cred(*krb5_context context*, *krb5_ccache id*, *krb5_flags which*,
*krb5_creds *cred*);

krb5_error_code
krb5_cc_set_default_name(*krb5_context context*, *const char *name*);

krb5_error_code
krb5_cc_start_seq_get(*krb5_context context*, *const krb5_ccache id*,
*krb5_cc_cursor *cursor*);

krb5_error_code
krb5_cc_store_cred(*krb5_context context*, *krb5_ccache id*, *krb5_creds *creds*);

krb5_error_code
krb5_cc_set_flags(*krb5_context context*, *krb5_cc_set_flags id*,
krb5_flags flags);

krb5_error_code
krb5_cc_next_cred(*krb5_context context*, *const krb5_ccache id*,
*krb5_cc_cursor *cursor*, *krb5_creds *creds*);

krb5_error_code
krb5_cc_next_cred_match(*krb5_context context*, *const krb5_ccache id*,
*krb5_cc_cursor *cursor*, *krb5_creds *creds*, *krb5_flags whichfields*,
*const krb5_creds *mcreds*);

krb5_error_code
krb5_cc_new_unique(*krb5_context context*, *const char *type*, *const char *hint*, *krb5_ccache *id*);

DESCRIPTION

The krb5_ccache structure holds a Kerberos credential cache.

The krb5_cc_cursor structure holds current position in a credential cache when iterating over the cache.

The krb5_cc_ops structure holds a set of operations that can me preformed on a credential cache.

There is no component inside krb5_ccache, krb5_cc_cursor nor krb5_fcc_ops that is directly referable.

The krb5_creds holds a Kerberos credential, see manpage for krb5_creds(3).

krb5_cc_default_name() and **krb5_cc_set_default_name**() gets and sets the default name for the *context*.

krb5_cc_default() opens the default credential cache in *id*. Return 0 or an error code.

krb5_cc_gen_new() generates a new credential cache of type *ops* in *id*. Return 0 or an error code. The Heimdal version of this function also runs **krb5_cc_initialize**() on the credential cache, but since the MIT version doesn't, portable code must call krb5_cc_initialize.

krb5_cc_new_unique() generates a new unique credential cache of *type* in *id*. If type is NULL, the library chooses the default credential cache type. The supplied *hint* (that can be NULL) is a string that the credential cache type can use to base the name of the credential on, this is to make it easier for the user to differentiate the credentials. The returned credential cache *id* should be freed using **krb5_cc_close**() or **krb5_cc_destroy**(). Returns 0 or an error code.

krb5_cc_resolve() finds and allocates a credential cache in *id* from the specification in *residual*. If the credential cache name doesn't contain any colon (:), interpret it as a file name. Return 0 or an error code.

krb5_cc_initialize() creates a new credential cache in *id* for *primary_principal*. Return 0 or an error code.

krb5_cc_close() stops using the credential cache *id* and frees the related resources. Return 0 or an error code. **krb5_cc_destroy**() removes the credential cache and closes (by calling **krb5_cc_close**()) *id*. Return 0 or an error code.

krb5_cc_copy_cache() copys the contents of *from* to *to*.

krb5_cc_get_full_name() returns the complete resolvable name of the credential cache *id* in *str*. *str* should be freed with free(3). Returns 0 or an error, on error *str is set to NULL.

krb5_cc_get_name() returns the name of the credential cache *id*.

krb5_cc_get_principal() returns the principal of *id* in *principal*. Return 0 or an error code.

krb5_cc_get_type() returns the type of the credential cache *id*.

krb5_cc_get_ops() returns the ops of the credential cache *id*.

krb5_cc_get_version() returns the version of *id*.

krb5_cc_register() Adds a new credential cache type with operations *ops*, overwriting any existing one if *override*. Return an error code or 0.

krb5_cc_get_prefix_ops() Get the cc ops that is registered in *context* to handle the *prefix*. Returns NULL if ops not found.

krb5_cc_remove_cred() removes the credential identified by (*cred*, *which*) from *id*.

krb5_cc_store_cred() stores *creds* in the credential cache *id*. Return 0 or an error code.

krb5_cc_set_flags() sets the flags of *id* to *flags*.

krb5_cc_clear_mcred() clears the *mcreds* argument so it is reset and can be used with *krb5_cc_retrieve_cred*.

krb5_cc_retrieve_cred(), retrieves the credential identified by *mcreds* (and *whichfields*) from *id* in *creds*. *creds* should be freed using **krb5_free_cred_contents**(). Return 0 or an error code.

krb5_cc_start_seq_get() initiates the krb5_cc_cursor structure to be used for iteration over the credential cache.

krb5_cc_next_cred() retrieves the next cred pointed to by (*id*, *cursor*) in *creds*, and advance *cursor*. Return 0 or an error code.

krb5_cc_next_cred_match() is similar to **krb5_cc_next_cred**() except that it will only return creds matching *whichfields* and *mcreds* (as interpreted by krb5_compare_creds(3).)

krb5_cc_end_seq_get() Destroys the cursor *cursor*.

EXAMPLE

This is a minimalistic version of **klist**.

```
#include <krb5/krb5.h>

int
main (int argc, char **argv)
{
    krb5_context context;
    krb5_cc_cursor cursor;
    krb5_error_code ret;
    krb5_ccache id;
    krb5_creds creds;

    if (krb5_init_context (&context) != 0)
        errx(1, "krb5_context");

    ret = krb5_cc_default (context, &id);
    if (ret)
        krb5_err(context, 1, ret, "krb5_cc_default");

    ret = krb5_cc_start_seq_get(context, id, &cursor);
    if (ret)
        krb5_err(context, 1, ret, "krb5_cc_start_seq_get");

    while((ret = krb5_cc_next_cred(context, id, &cursor, &creds)) == 0){
        char *principal;

        krb5_unparse_name_short(context, creds.server, &principal);
        printf("principal: %s\n", principal);
        free(principal);
        krb5_free_cred_contents (context, &creds);
```

```
        }
        ret = krb5_cc_end_seq_get(context, id, &cursor);
        if (ret)
            krb5_err(context, 1, ret, "krb5_cc_end_seq_get");

        krb5_cc_close(context, id);

        krb5_free_context(context);
        return 0;
    }
```

SEE ALSO

krb5(3), krb5.conf(5), kerberos(8)

NAME

krb5_check_transited, **krb5_check_transited_realms**, **krb5_domain_x500_decode**, **krb5_domain_x500_encode** — realm transit verification and encoding/decoding functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_check_transited(krb5_context context, krb5_const_realm client_realm,
    krb5_const_realm server_realm, krb5_realm *realms, int num_realms,
    int *bad_realm);

krb5_error_code
krb5_check_transited_realms(krb5_context context,
    const char *const *realms, int num_realms, int *bad_realm);

krb5_error_code
krb5_domain_x500_decode(krb5_context context, krb5_data tr, char ***realms,
    int *num_realms, const char *client_realm, const char *server_realm);

krb5_error_code
krb5_domain_x500_encode(char **realms, int num_realms, krb5_data *encoding);
```

DESCRIPTION

krb5_check_transited() checks the path from *client_realm* to *server_realm* where *realms* and *num_realms* is the realms between them. If the function returns an error value, *bad_realm* will be set to the realm in the list causing the error. **krb5_check_transited**() is used internally by the KDC and libkrb5 and should not be called by client applications.

krb5_check_transited_realms() is deprecated.

krb5_domain_x500_encode() and **krb5_domain_x500_decode**() encodes and decodes the realm names in the X500 format that Kerberos uses to describe the transited realms in krbtgts.

SEE ALSO

krb5(3), krb5.conf(5)

NAME

krb5_compare_creds — compare Kerberos 5 credentials

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_boolean
krb5_compare_creds(krb5_context context, krb5_flags whichfields,
    const krb5_creds *mcreds, const krb5_creds *creds);
```

DESCRIPTION

krb5_compare_creds() compares *mcreds* (usually filled in by the application) to *creds* (most often from a credentials cache) and return TRUE if they are equal. Unless *mcreds->server* is NULL, the service of the credentials are always compared. If the client name in *mcreds* is present, the client names are also compared. This function is normally only called indirectly via krb5_cc_retrieve_cred(3).

The following flags, set in *whichfields*, affects the comparison:

KRB5_TC_MATCH_SRV_NAMEONLY	Consider all realms equal when comparing the service principal.
KRB5_TC_MATCH_KEYTYPE	Compare enctypes.
KRB5_TC_MATCH_FLAGS_EXACT	Make sure that the ticket flags are identical.
KRB5_TC_MATCH_FLAGS	Make sure that all ticket flags set in *mcreds* are also present in *creds*.
KRB5_TC_MATCH_TIMES_EXACT	Compares the ticket times exactly.
KRB5_TC_MATCH_TIMES	Compares only the expiration times of the creds.
KRB5_TC_MATCH_AUTHDATA	Compares the authdata fields.
KRB5_TC_MATCH_2ND_TKT	Compares the second tickets (used by user-to-user authentication).
KRB5_TC_MATCH_IS_SKEY	Compares the existance of the second ticket.

SEE ALSO

krb5(3), krb5_cc_retrieve_cred(3), krb5_creds(3), krb5_get_init_creds(3), kerberos(8)

NAME

krb5_config_file_free, **krb5_config_free_strings**, **krb5_config_get**, **krb5_config_get_bool**, **krb5_config_get_bool_default**, **krb5_config_get_int**, **krb5_config_get_int_default**, **krb5_config_get_list**, **krb5_config_get_next**, **krb5_config_get_string**, **krb5_config_get_string_default**, **krb5_config_get_strings**, **krb5_config_get_time**, **krb5_config_get_time_default**, **krb5_config_parse_file**, **krb5_config_parse_file_multi**, **krb5_config_vget**, **krb5_config_vget_bool**, **krb5_config_vget_bool_default**, **krb5_config_vget_int**, **krb5_config_vget_int_default**, **krb5_config_vget_list**, **krb5_config_vget_next**, **krb5_config_vget_string**, **krb5_config_vget_string_default**, **krb5_config_vget_strings**, **krb5_config_vget_time**, **krb5_config_vget_time_default** — get configuration value

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_config_file_free(krb5_context context, krb5_config_section *s);

void
krb5_config_free_strings(char **strings);

const void *
krb5_config_get(krb5_context context, const krb5_config_section *c,
    int type, ...);

krb5_boolean
krb5_config_get_bool(krb5_context context, krb5_config_section *c, ...);

krb5_boolean
krb5_config_get_bool_default(krb5_context context, krb5_config_section *c,
    krb5_boolean def_value, ...);

int
krb5_config_get_int(krb5_context context, krb5_config_section *c, ...);

int
krb5_config_get_int_default(krb5_context context, krb5_config_section *c,
    int def_value, ...);

const char*
krb5_config_get_string(krb5_context context, krb5_config_section *c, ...);

const char*
krb5_config_get_string_default(krb5_context context,
    krb5_config_section *c, const char *def_value, ...);

char**
krb5_config_get_strings(krb5_context context,
    const krb5_config_section *c, ...);

int
krb5_config_get_time(krb5_context context, krb5_config_section *c, ...);
```

int
krb5_config_get_time_default(*krb5_context context*, *krb5_config_section *c*, *int def_value*, *...*);

krb5_error_code
krb5_config_parse_file(*krb5_context context*, *const char *fname*, *krb5_config_section **res*);

krb5_error_code
krb5_config_parse_file_multi(*krb5_context context*, *const char *fname*, *krb5_config_section **res*);

*const void **
krb5_config_vget(*krb5_context context*, *const krb5_config_section *c*, *int type*, *va_list args*);

krb5_boolean
krb5_config_vget_bool(*krb5_context context*, *const krb5_config_section *c*, *va_list args*);

krb5_boolean
krb5_config_vget_bool_default(*krb5_context context*, *const krb5_config_section *c*, *krb5_boolean def_value*, *va_list args*);

int
krb5_config_vget_int(*krb5_context context*, *const krb5_config_section *c*, *va_list args*);

int
krb5_config_vget_int_default(*krb5_context context*, *const krb5_config_section *c*, *int def_value*, *va_list args*);

*const krb5_config_binding **
krb5_config_vget_list(*krb5_context context*, *const krb5_config_section *c*, *va_list args*);

*const void **
krb5_config_vget_next(*krb5_context context*, *const krb5_config_section *c*, *const krb5_config_binding **pointer*, *int type*, *va_list args*);

*const char **
krb5_config_vget_string(*krb5_context context*, *const krb5_config_section *c*, *va_list args*);

*const char **
krb5_config_vget_string_default(*krb5_context context*, *const krb5_config_section *c*, *const char *def_value*, *va_list args*);

*char ***
krb5_config_vget_strings(*krb5_context context*, *const krb5_config_section *c*, *va_list args*);

int
krb5_config_vget_time(*krb5_context context*, *const krb5_config_section *c*, *va_list args*);

int
krb5_config_vget_time_default(*krb5_context context*, *const krb5_config_section *c*, *int def_value*, *va_list args*);

DESCRIPTION

These functions get values from the krb5.conf(5) configuration file, or another configuration database specified by the *c* parameter.

The variable arguments should be a list of strings naming each subsection to look for. For example:

```
krb5_config_get_bool_default(context, NULL, FALSE,
     "libdefaults", "log_utc", NULL);
```

gets the boolean value for the log_utc option, defaulting to FALSE.

krb5_config_get_bool_default() will convert the option value to a boolean value, where 'yes', 'true', and any non-zero number means TRUE, and any other value FALSE.

krb5_config_get_int_default() will convert the value to an integer.

krb5_config_get_time_default() will convert the value to a period of time (not a time stamp) in seconds, so the string '2 weeks' will be converted to 1209600 ($2 * 7 * 24 * 60 * 60$).

krb5_config_get_string() returns a *const char* * to a string in the configuration database. The string not be valid after reload of the configuration database so a caller should make a local copy if its need to keep the database.

krb5_config_free_strings() free *strings* as returned by **krb5_config_get_strings**() and **krb5_config_vget_strings**(). If the argument *strings* is a NULL pointer, no action occurs.

krb5_config_file_free() free the result of **krb5_config_parse_file**() and **krb5_config_parse_file_multi**().

SEE ALSO

krb5_appdefault(3), krb5_init_context(3), krb5.conf(5)

BUGS

For the default functions, other than for the string case, there's no way to tell whether there was a value specified or not.

NAME

krb5_context — krb5 state structure

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>
```

DESCRIPTION

The **krb5_context** structure is designed to hold all per thread state. All global variables that are context specific are stored in this structure, including default encryption types, credentials-cache (ticket file), and default realms.

The internals of the structure should never be accessed directly, functions exist for extracting information.

SEE ALSO

krb5_init_context(3), kerberos(8)

NAME

krb5_checksum, **krb5_checksum_disable**, **krb5_checksum_is_collision_proof**, **krb5_checksum_is_keyed**, **krb5_checksumsize**, **krb5_cksumtype_valid**, **krb5_copy_checksum**, **krb5_create_checksum**, **krb5_crypto_get_checksum_type** **krb5_free_checksum**, **krb5_free_checksum_contents**, **krb5_hmac**, **krb5_verify_checksum** — creates, handles and verifies checksums

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

typedef Checksum krb5_checksum;

void
krb5_checksum_disable(krb5_context context, krb5_cksumtype type);

krb5_boolean
krb5_checksum_is_collision_proof(krb5_context context,
    krb5_cksumtype type);

krb5_boolean
krb5_checksum_is_keyed(krb5_context context, krb5_cksumtype type);

krb5_error_code
krb5_cksumtype_valid(krb5_context context, krb5_cksumtype ctype);

krb5_error_code
krb5_checksumsize(krb5_context context, krb5_cksumtype type, size_t *size);

krb5_error_code
krb5_create_checksum(krb5_context context, krb5_crypto crypto,
    krb5_key_usage usage, int type, void *data, size_t len,
    Checksum *result);

krb5_error_code
krb5_verify_checksum(krb5_context context, krb5_crypto crypto,
    krb5_key_usage usage, void *data, size_t len, Checksum *cksum);

krb5_error_code
krb5_crypto_get_checksum_type(krb5_context context, krb5_crypto crypto,
    krb5_cksumtype *type);

void
krb5_free_checksum(krb5_context context, krb5_checksum *cksum);

void
krb5_free_checksum_contents(krb5_context context, krb5_checksum *cksum);

krb5_error_code
krb5_hmac(krb5_context context, krb5_cksumtype cktype, const void *data,
    size_t len, unsigned usage, krb5_keyblock *key, Checksum *result);

krb5_error_code
krb5_copy_checksum(krb5_context context, const krb5_checksum *old,
    krb5_checksum **new);
```

DESCRIPTION

The `krb5_checksum` structure holds a Kerberos checksum. There is no component inside `krb5_checksum` that is directly referable.

The functions are used to create and verify checksums. **krb5_create_checksum**() creates a checksum of the specified data, and puts it in *result*. If *crypto* is `NULL`, *usage_or_type* specifies the checksum type to use; it must not be keyed. Otherwise *crypto* is an encryption context created by **krb5_crypto_init**(), and *usage_or_type* specifies a key-usage.

krb5_verify_checksum() verifies the *checksum* against the provided data.

krb5_checksum_is_collision_proof() returns true is the specified checksum is collision proof (that it's very unlikely that two strings has the same hash value, and that it's hard to find two strings that has the same hash). Examples of collision proof checksums are MD5, and SHA1, while CRC32 is not.

krb5_checksum_is_keyed() returns true if the specified checksum type is keyed (that the hash value is a function of both the data, and a separate key). Examples of keyed hash algorithms are HMAC-SHA1-DES3, and RSA-MD5-DES. The "plain" hash functions MD5, and SHA1 are not keyed.

krb5_crypto_get_checksum_type() returns the checksum type that will be used when creating a checksum for the given *crypto* context. This function is useful in combination with **krb5_checksumsize**() when you want to know the size a checksum will use when you create it.

krb5_cksumtype_valid() returns 0 or an error if the checksumtype is implemented and not currently disabled in this kerberos library.

krb5_checksumsize() returns the size of the outdata of checksum function.

krb5_copy_checksum() returns a copy of the checksum **krb5_free_checksum**() should use used to free the *new* checksum.

krb5_free_checksum() free the checksum and the content of the checksum.

krb5_free_checksum_contents() frees the content of checksum in *cksum*.

krb5_hmac() calculates the HMAC over *data* (with length *len*) using the keyusage *usage* and keyblock *key*. Note that keyusage is not always used in checksums.

krb5_checksum_disable globally disables the checksum type.

SEE ALSO

`krb5_crypto_init`(3), `krb5_c_encrypt`(3), `krb5_encrypt`(3)

NAME

krb5_creds, **krb5_copy_creds**, **krb5_copy_creds_contents**, **krb5_free_creds**, **krb5_free_cred_contents** — Kerberos 5 credential handling functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_copy_creds(krb5_context context, const krb5_creds *incred,
    krb5_creds **outcred);

krb5_error_code
krb5_copy_creds_contents(krb5_context context, const krb5_creds *incred,
    krb5_creds *outcred);

krb5_error_code
krb5_free_creds(krb5_context context, krb5_creds *outcred);

krb5_error_code
krb5_free_cred_contents(krb5_context context, krb5_creds *cred);
```

DESCRIPTION

krb5_creds holds Kerberos credentials:

```
typedef struct krb5_creds {
    krb5_principal    client;
    krb5_principal    server;
    krb5_keyblock     session;
    krb5_times        times;
    krb5_data         ticket;
    krb5_data         second_ticket;
    krb5_authdata     authdata;
    krb5_addresses    addresses;
    krb5_ticket_flags flags;
} krb5_creds;
```

krb5_copy_creds() makes a copy of *incred* to *outcred*. *outcred* should be freed with **krb5_free_creds**() by the caller.

krb5_copy_creds_contents() makes a copy of the content of *incred* to *outcreds*. *outcreds* should be freed by the called with **krb5_free_creds_contents**().

krb5_free_creds() frees the content of the *cred* structure and the structure itself.

krb5_free_cred_contents() frees the content of the *cred* structure.

SEE ALSO

krb5(3), krb5_compare_creds(3), krb5_get_init_creds(3), kerberos(8)

NAME

krb5_crypto_destroy, **krb5_crypto_init** — encryption support in krb5

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_crypto_init(krb5_context context, krb5_keyblock *key,
    krb5_enctype enctype, krb5_crypto *crypto);

krb5_error_code
krb5_crypto_destroy(krb5_context context, krb5_crypto crypto);
```

DESCRIPTION

Heimdal exports parts of the Kerberos crypto interface for applications.

Each kerberos encrytion/checksum function takes a crypto context.

To setup and destroy crypto contextes there are two functions **krb5_crypto_init**() and **krb5_crypto_destroy**(). The encryption type to use is taken from the key, but can be overridden with the *enctype parameter*. This can be useful for encryptions types which is compatiable (DES for example).

SEE ALSO

krb5_create_checksum(3), krb5_encrypt(3)

NAME

krb5_data, **krb5_data_zero**, **krb5_data_free**, **krb5_free_data_contents**, **krb5_free_data**, **krb5_data_alloc**, **krb5_data_realloc**, **krb5_data_copy**, **krb5_copy_data**, **krb5_data_cmp** — operates on the Kerberos datatype krb5_data

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

struct krb5_data;

void
krb5_data_zero(krb5_data *p);

void
krb5_data_free(krb5_data *p);

void
krb5_free_data_contents(krb5_context context, krb5_data *p);

void
krb5_free_data(krb5_context context, krb5_data *p);

krb5_error_code
krb5_data_alloc(krb5_data *p, int len);

krb5_error_code
krb5_data_realloc(krb5_data *p, int len);

krb5_error_code
krb5_data_copy(krb5_data *p, const void *data, size_t len);

krb5_error_code
krb5_copy_data(krb5_context context, const krb5_data *indata,
    krb5_data **outdata);

krb5_error_code
krb5_data_cmp(const krb5_data *data1, const krb5_data *data2);
```

DESCRIPTION

The krb5_data structure holds a data element. The structure contains two public accessible elements *length* (the length of data) and *data* (the data itself). The structure must always be initiated and freed by the functions documented in this manual.

krb5_data_zero() resets the content of *p*.

krb5_data_free() free the data in *p* and reset the content of the structure with **krb5_data_zero**().

krb5_free_data_contents() works the same way as *krb5_data_free*. The diffrence is that krb5_free_data_contents is more portable (exists in MIT api).

krb5_free_data() frees the data in *p* and *p* itself.

krb5_data_alloc() allocates *len* bytes in *p*. Returns 0 or an error.

krb5_data_realloc() reallocates the length of *p* to the length in *len*. Returns 0 or an error.

krb5_data_copy() copies the *data* that have the length *len* into *p*. *p* is not freed so the calling function should make sure the *p* doesn't contain anything needs to be freed. Returns 0 or an error.

krb5_copy_data() copies the krb5_data in *indata* to *outdata*. *outdata* is not freed so the calling function should make sure the *outdata* doesn't contain anything needs to be freed. *outdata* should be freed using **krb5_free_data**(). Returns 0 or an error.

krb5_data_cmp() will compare two data object and check if they are the same in a simular way as memcmp does it. The return value can be used for sorting.

SEE ALSO

krb5(3), krb5_storage(3), kerberos(8)

NAME

krb5_digest, **krb5_digest_alloc**, **krb5_digest_free**, **krb5_digest_set_server_cb**, **krb5_digest_set_type**, **krb5_digest_set_hostname**, **krb5_digest_get_server_nonce**, **krb5_digest_set_server_nonce**, **krb5_digest_get_opaque**, **krb5_digest_set_opaque**, **krb5_digest_get_identifier**, **krb5_digest_set_identifier**, **krb5_digest_init_request**, **krb5_digest_set_client_nonce**, **krb5_digest_set_digest**, **krb5_digest_set_username**, **krb5_digest_set_authid**, **krb5_digest_set_authentication_user**, **krb5_digest_set_realm**, **krb5_digest_set_method**, **krb5_digest_set_uri**, **krb5_digest_set_nonceCount**, **krb5_digest_set_qop**, **krb5_digest_request**, **krb5_digest_get_responseData**, **krb5_digest_get_rsp**, **krb5_digest_get_tickets**, **krb5_digest_get_client_binding**, **krb5_digest_get_a1_hash** — remote digest (HTTP-DIGEST, SASL, CHAP) suppport

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

typedef struct krb5_digest *krb5_digest;

krb5_error_code
krb5_digest_alloc(krb5_context context, krb5_digest *digest);

void
krb5_digest_free(krb5_digest digest);

krb5_error_code
krb5_digest_set_type(krb5_context context, krb5_digest digest,
    const char *type);

krb5_error_code
krb5_digest_set_server_cb(krb5_context context, krb5_digest digest,
    const char *type, const char *binding);

krb5_error_code
krb5_digest_set_hostname(krb5_context context, krb5_digest digest,
    const char *hostname);

const char *
krb5_digest_get_server_nonce(krb5_context context, krb5_digest digest);

krb5_error_code
krb5_digest_set_server_nonce(krb5_context context, krb5_digest digest,
    const char *nonce);

const char *
krb5_digest_get_opaque(krb5_context context, krb5_digest digest);

krb5_error_code
krb5_digest_set_opaque(krb5_context context, krb5_digest digest,
    const char *opaque);

const char *
krb5_digest_get_identifier(krb5_context context, krb5_digest digest);
```

krb5_error_code
krb5_digest_set_identifier(*krb5_context context*, *krb5_digest digest*, *const char *id*);

krb5_error_code
krb5_digest_init_request(*krb5_context context*, *krb5_digest digest*, *krb5_realm realm*, *krb5_ccache ccache*);

krb5_error_code
krb5_digest_set_client_nonce(*krb5_context context*, *krb5_digest digest*, *const char *nonce*);

krb5_error_code
krb5_digest_set_digest(*krb5_context context*, *krb5_digest digest*, *const char *dgst*);

krb5_error_code
krb5_digest_set_username(*krb5_context context*, *krb5_digest digest*, *const char *username*);

krb5_error_code
krb5_digest_set_authid(*krb5_context context*, *krb5_digest digest*, *const char *authid*);

krb5_error_code
krb5_digest_set_authentication_user(*krb5_context context*, *krb5_digest digest*, *krb5_principal authentication_user*);

krb5_error_code
krb5_digest_set_realm(*krb5_context context*, *krb5_digest digest*, *const char *realm*);

krb5_error_code
krb5_digest_set_method(*krb5_context context*, *krb5_digest digest*, *const char *method*);

krb5_error_code
krb5_digest_set_uri(*krb5_context context*, *krb5_digest digest*, *const char *uri*);

krb5_error_code
krb5_digest_set_nonceCount(*krb5_context context*, *krb5_digest digest*, *const char *nonce_count*);

krb5_error_code
krb5_digest_set_qop(*krb5_context context*, *krb5_digest digest*, *const char *qop*);

krb5_error_code
krb5_digest_request(*krb5_context context*, *krb5_digest digest*, *krb5_realm realm*, *krb5_ccache ccache*);

*const char **
krb5_digest_get_responseData(*krb5_context context*, *krb5_digest digest*);

*const char **
krb5_digest_get_rsp(*krb5_context context*, *krb5_digest digest*);

krb5_error_code
krb5_digest_get_tickets(*krb5_context context*, *krb5_digest digest*, *Ticket **tickets*);

krb5_error_code
krb5_digest_get_client_binding(*krb5_context context*, *krb5_digest digest*, *char **type*, *char **binding*);

krb5_error_code
krb5_digest_get_a1_hash(*krb5_context context*, *krb5_digest digest*, *krb5_data *data*);

DESCRIPTION

The **krb5_digest_alloc**() function allocatates the *digest* structure. The structure should be freed with **krb5_digest_free**() when it is no longer being used.

krb5_digest_alloc() returns 0 to indicate success. Otherwise an kerberos code is returned and the pointer that *digest* points to is set to NULL.

krb5_digest_free() free the structure *digest*.

SEE ALSO

krb5(3), kerberos(8)

NAME

krb5_eai_to_heim_errno, **krb5_h_errno_to_heim_errno** — convert resolver error code to com_err error codes

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_eai_to_heim_errno(int eai_errno, int system_error);

krb5_error_code
krb5_h_errno_to_heim_errno(int eai_errno);
```

DESCRIPTION

krb5_eai_to_heim_errno() and **krb5_h_errno_to_heim_errno**() convert getaddrinfo(3), getnameinfo(3), and h_errno(3) to com_err error code that are used by Heimdal, this is useful for for function returning kerberos errors and needs to communicate failures from resolver function.

SEE ALSO

krb5(3), kerberos(8)

NAME

krb5_crypto_getblocksize, **krb5_crypto_getconfoundersize** **krb5_crypto_getenctype**, **krb5_crypto_getpadsize**, **krb5_crypto_overhead**, **krb5_decrypt**, **krb5_decrypt_EncryptedData**, **krb5_decrypt_ivec**, **krb5_decrypt_ticket**, **krb5_encrypt**, **krb5_encrypt_EncryptedData**, **krb5_encrypt_ivec**, **krb5_enctype_disable**, **krb5_enctype_keysize**, **krb5_enctype_to_string**, **krb5_enctype_valid**, **krb5_get_wrapped_length**, **krb5_string_to_enctype** — encrypt and decrypt data, set and get encryption type parameters

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_encrypt(krb5_context context, krb5_crypto crypto, unsigned usage,
    void *data, size_t len, krb5_data *result);

krb5_error_code
krb5_encrypt_EncryptedData(krb5_context context, krb5_crypto crypto,
    unsigned usage, void *data, size_t len, int kvno, EncryptedData *result);

krb5_error_code
krb5_encrypt_ivec(krb5_context context, krb5_crypto crypto, unsigned usage,
    void *data, size_t len, krb5_data *result, void *ivec);

krb5_error_code
krb5_decrypt(krb5_context context, krb5_crypto crypto, unsigned usage,
    void *data, size_t len, krb5_data *result);

krb5_error_code
krb5_decrypt_EncryptedData(krb5_context context, krb5_crypto crypto,
    unsigned usage, EncryptedData *e, krb5_data *result);

krb5_error_code
krb5_decrypt_ivec(krb5_context context, krb5_crypto crypto, unsigned usage,
    void *data, size_t len, krb5_data *result, void *ivec);

krb5_error_code
krb5_decrypt_ticket(krb5_context context, Ticket *ticket,
    krb5_keyblock *key, EncTicketPart *out, krb5_flags flags);

krb5_error_code
krb5_crypto_getblocksize(krb5_context context, size_t *blocksize);

krb5_error_code
krb5_crypto_getenctype(krb5_context context, krb5_crypto crypto,
    krb5_enctype *enctype);

krb5_error_code
krb5_crypto_getpadsize(krb5_context context, size_t, *padsize");

krb5_error_code
krb5_crypto_getconfoundersize(krb5_context context, krb5_crypto crypto,
    size_t, *confoundersize");
```

krb5_error_code
krb5_enctype_keysize(*krb5_context context*, *krb5_enctype type*, *size_t *keysize*);

krb5_error_code
krb5_crypto_overhead(*krb5_context context*, *size_t*, *padsize"*);

krb5_error_code
krb5_string_to_enctype(*krb5_context context*, *const char *string*, *krb5_enctype *etype*);

krb5_error_code
krb5_enctype_to_string(*krb5_context context*, *krb5_enctype etype*, *char **string*);

krb5_error_code
krb5_enctype_valid(*krb5_context context*, *krb5_enctype etype*);

void
krb5_enctype_disable(*krb5_context context*, *krb5_enctype etype*);

size_t
krb5_get_wrapped_length(*krb5_context context*, *krb5_crypto crypto*, *size_t data_len*);

DESCRIPTION

These functions are used to encrypt and decrypt data.

krb5_encrypt_ivec() puts the encrypted version of *data* (of size *len*) in *result*. If the encryption type supports using derived keys, *usage* should be the appropriate key-usage. *ivec* is a pointer to a initial IV, it is modified to the end IV at the end of the round. Ivec should be the size of If NULL is passed in, the default IV is used. **krb5_encrypt**() does the same as **krb5_encrypt_ivec**() but with *ivec* being NULL. **krb5_encrypt_EncryptedData**() does the same as **krb5_encrypt**(), but it puts the encrypted data in a *EncryptedData* structure instead. If *kvno* is not zero, it will be put in the (optional) *kvno* field in the *EncryptedData*.

krb5_decrypt_ivec(), **krb5_decrypt**(), and **krb5_decrypt_EncryptedData**() works similarly.

krb5_decrypt_ticket() decrypts the encrypted part of *ticket* with *key*. **krb5_decrypt_ticket**() also verifies the timestamp in the ticket, invalid flag and if the KDC haven't verified the transited path, the transit path.

krb5_enctype_keysize(), **krb5_crypto_getconfoundersize**(), **krb5_crypto_getblocksize**(), **krb5_crypto_getenctype**(), **krb5_crypto_getpadsize**(), **krb5_crypto_overhead**() all returns various (sometimes) useful information from a crypto context. **krb5_crypto_overhead**() is the combination of krb5_crypto_getconfoundersize, krb5_crypto_getblocksize and krb5_crypto_getpadsize and return the maximum overhead size.

krb5_enctype_to_string() converts a encryption type number to a string that can be printable and stored. The strings returned should be freed with free(3).

krb5_string_to_enctype() converts a encryption type strings to a encryption type number that can use used for other Kerberos crypto functions.

krb5_enctype_valid() returns 0 if the encrypt is supported and not disabled, otherwise and error code is returned.

krb5_enctype_disable() (globally, for all contextes) disables the *enctype*.

krb5_get_wrapped_length() returns the size of an encrypted packet by *crypto* of length *data_len*.

SEE ALSO

krb5_create_checksum(3), krb5_crypto_init(3)

NAME

krb5_expand_hostname, **krb5_expand_hostname_realms** — Kerberos 5 host name canonicalization functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_expand_hostname(krb5_context context, const char *orig_hostname,
    char **new_hostname);

krb5_error_code
krb5_expand_hostname_realms(krb5_context context,
    const char *orig_hostname, char **new_hostname, char ***realms);
```

DESCRIPTION

krb5_expand_hostname() tries to make *orig_hostname* into a more canonical one in the newly allocated space returned in *new_hostname*. Caller must free the hostname with free(3).

krb5_expand_hostname_realms() expands *orig_hostname* to a name we believe to be a hostname in newly allocated space in *new_hostname* and return the realms *new_hostname* is belive to belong to in *realms*. *Realms* is a array terminated with NULL. Caller must free the *realms* with **krb5_free_host_realm**() and *new_hostname* with free(3).

SEE ALSO

krb5(3), krb5_free_host_realm(3), krb5_get_host_realm(3), kerberos(8)

NAME

krb5_find_padata, **krb5_padata_add** — Kerberos 5 pre-authentication data handling functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

PA_DATA *
krb5_find_padata(PA_DATA *val, unsigned len, int type, int *index);

int
krb5_padata_add(krb5_context context, METHOD_DATA *md, int type, void *buf,
    size_t len);
```

DESCRIPTION

krb5_find_padata() tries to find the pre-authentication data entry of type *type* in the array *val* of length *len*. The search is started at entry pointed out by **index* (zero based indexing). If the type isn't found, NULL is returned.

krb5_padata_add() adds a pre-authentication data entry of type *type* pointed out by *buf* and *len* to *md*.

SEE ALSO

krb5(3), kerberos(8)

NAME

krb5_generate_random_block — Kerberos 5 random functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

void
krb5_generate_random_block(void *buf, size_t len);
```

DESCRIPTION

krb5_generate_random_block() generates a cryptographically strong pseudo-random block into the buffer *buf* of length *len*.

SEE ALSO

krb5(3), krb5.conf(5)

NAME

krb5_get_all_client_addrs, **krb5_get_all_server_addrs** — return local addresses

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_get_all_client_addrs(krb5_context context, krb5_addresses *addrs);

krb5_error_code
krb5_get_all_server_addrs(krb5_context context, krb5_addresses *addrs);
```

DESCRIPTION

These functions return in *addrs* a list of addresses associated with the local host.

The server variant returns all configured interface addresses (if possible), including loop-back addresses. This is useful if you want to create sockets to listen to.

The client version will also scan local interfaces (can be turned off by setting `libdefaults/scan_interfaces` to false in `krb5.conf`), but will not include loop-back addresses, unless there are no other addresses found. It will remove all addresses included in `libdefaults/ignore_addresses` but will unconditionally include addresses in `libdefaults/extra_addresses`.

The returned addresses should be freed by calling **krb5_free_addresses**().

SEE ALSO

krb5_free_addresses(3)

NAME

krb5_get_credentials, **krb5_get_credentials_with_flags**, **krb5_get_cred_from_kdc**, **krb5_get_cred_from_kdc_opt**, **krb5_get_kdc_cred**, **krb5_get_renewed_creds** — get credentials from the KDC using krbtgt

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_get_credentials(krb5_context context, krb5_flags options,
    krb5_ccache ccache, krb5_creds *in_creds, krb5_creds **out_creds);

krb5_error_code
krb5_get_credentials_with_flags(krb5_context context, krb5_flags options,
    krb5_kdc_flags flags, krb5_ccache ccache, krb5_creds *in_creds,
    krb5_creds **out_creds);

krb5_error_code
krb5_get_cred_from_kdc(krb5_context context, krb5_ccache ccache,
    krb5_creds *in_creds, krb5_creds **out_creds, krb5_creds ***ret_tgts);

krb5_error_code
krb5_get_cred_from_kdc_opt(krb5_context context, krb5_ccache ccache,
    krb5_creds *in_creds, krb5_creds **out_creds, krb5_creds ***ret_tgts,
    krb5_flags flags);

krb5_error_code
krb5_get_kdc_cred(krb5_context context, krb5_ccache id,
    krb5_kdc_flags flags, krb5_addresses *addresses, Ticket *second_ticket,
    krb5_creds *in_creds, krb5_creds **out_creds);

krb5_error_code
krb5_get_renewed_creds(krb5_context context, krb5_creds *creds,
    krb5_const_principal client, krb5_ccache ccache,
    const char *in_tkt_service);
```

DESCRIPTION

krb5_get_credentials_with_flags() get credentials specified by *in_creds->server* and *in_creds->client* (the rest of the *in_creds* structure is ignored) by first looking in the *ccache* and if doesn't exists or is expired, fetch the credential from the KDC using the krbtgt in *ccache*. The credential is returned in *out_creds* and should be freed using the function **krb5_free_creds**().

Valid flags to pass into *options* argument are:

KRB5_GC_CACHED Only check the *ccache*, don't got out on network to fetch credential.

KRB5_GC_USER_USER
: Request a user to user ticket. This option doesn't store the resulting user to user credential in the *ccache*.

KRB5_GC_EXPIRED_OK
: returns the credential even if it is expired, default behavior is trying to refetch the credential from the KDC.

Flags are KDCOptions, note the caller must fill in the bit-field and not use the integer associated structure.

krb5_get_credentials() works the same way as **krb5_get_credentials_with_flags**() except that the *flags* field is missing.

krb5_get_cred_from_kdc() and **krb5_get_cred_from_kdc_opt**() fetches the credential from the KDC very much like **krb5_get_credentials,**(*but*, *doesn't*, *look*, *in*, *the*) *ccache* if the credential exists there first.

krb5_get_kdc_cred() does the same as the functions above, but the caller must fill in all the information andits closer to the wire protocol.

krb5_get_renewed_creds() renews a credential given by *in_tkt_service* (if NULL the default krbtgt) using the credential cache *ccache*. The result is stored in *creds* and should be freed using *krb5_free_creds*.

EXAMPLES

Here is a example function that get a credential from a credential cache *id* or the KDC and returns it to the caller.

```
#include <krb5/krb5.h>

int
getcred(krb5_context context, krb5_ccache id, krb5_creds **creds)
{
    krb5_error_code ret;
    krb5_creds in;

    ret = krb5_parse_name(context, "client@EXAMPLE.COM",
                          &in.client);
    if (ret)
        krb5_err(context, 1, ret, "krb5_parse_name");

    ret = krb5_parse_name(context, "host/server.example.com@EXAMPLE.COM",
                          &in.server);
    if (ret)
        krb5_err(context, 1, ret, "krb5_parse_name");

    ret = krb5_get_credentials(context, 0, id, &in, creds);
    if (ret)
        krb5_err(context, 1, ret, "krb5_get_credentials");

    return 0;
}
```

SEE ALSO

krb5(3), krb5_get_forwarded_creds(3), krb5.conf(5)

NAME

krb5_get_creds, **krb5_get_creds_opt_add_options**, **krb5_get_creds_opt_alloc**, **krb5_get_creds_opt_free**, **krb5_get_creds_opt_set_enctype**, **krb5_get_creds_opt_set_impersonate**, **krb5_get_creds_opt_set_options**, **krb5_get_creds_opt_set_ticket** — get credentials from the KDC

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_get_creds(krb5_context context, krb5_get_creds_opt opt,
    krb5_ccache ccache, krb5_const_principal inprinc,
    krb5_creds **out_creds);

void
krb5_get_creds_opt_add_options(krb5_context context,
    krb5_get_creds_opt opt, krb5_flags options);

krb5_error_code
krb5_get_creds_opt_alloc(krb5_context context, krb5_get_creds_opt *opt);

void
krb5_get_creds_opt_free(krb5_context context, krb5_get_creds_opt opt);

void
krb5_get_creds_opt_set_enctype(krb5_context context,
    krb5_get_creds_opt opt, krb5_enctype enctype);

krb5_error_code
krb5_get_creds_opt_set_impersonate(krb5_context context,
    krb5_get_creds_opt opt, krb5_const_principal self);

void
krb5_get_creds_opt_set_options(krb5_context context,
    krb5_get_creds_opt opt, krb5_flags options);

krb5_error_code
krb5_get_creds_opt_set_ticket(krb5_context context,
    krb5_get_creds_opt opt, const Ticket *ticket);
```

DESCRIPTION

krb5_get_creds() fetches credentials specified by *opt* by first looking in the *ccache*, and then it doesn't exists, fetch the credential from the KDC using the krbtgts in *ccache*. The credential is returned in *out_creds* and should be freed using the function **krb5_free_creds**().

The structure krb5_get_creds_opt controls the behavior of **krb5_get_creds**(). The structure is opaque to consumers that can set the content of the structure with accessors functions. All accessor functions make copies of the data that is passed into accessor functions, so external consumers free the memory before calling **krb5_get_creds**().

The structure krb5_get_creds_opt is allocated with **krb5_get_creds_opt_alloc**() and freed with **krb5_get_creds_opt_free**(). The free function also frees the content of the structure set by the accessor functions.

krb5_get_creds_opt_add_options() and **krb5_get_creds_opt_set_options**() adds and sets options to the structure . The possible options to set are

KRB5_GC_CACHED Only check the *ccache*, don't got out on network to fetch credential.

KRB5_GC_USER_USER
request a user to user ticket. This options doesn't store the resulting user to user credential in the *ccache*.

KRB5_GC_EXPIRED_OK
returns the credential even if it is expired, default behavior is trying to refetch the credential from the KDC.

KRB5_GC_NO_STORE Do not store the resulting credentials in the *ccache*.

krb5_get_creds_opt_set_enctype() sets the preferred encryption type of the application. Don't set this unless you have to since if there is no match in the KDC, the function call will fail.

krb5_get_creds_opt_set_impersonate() sets the principal to impersonate., Returns a ticket that have the impersonation principal as a client and the requestor as the service. Note that the requested principal have to be the same as the client principal in the krbtgt.

krb5_get_creds_opt_set_ticket() sets the extra ticket used in user-to-user or contrained delegation use case.

SEE ALSO

krb5(3), krb5_get_credentials(3), krb5.conf(5)

NAME

krb5_get_forwarded_creds, **krb5_fwd_tgt_creds** — get forwarded credentials from the KDC

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_get_forwarded_creds(krb5_context context,
    krb5_auth_context auth_context, krb5_ccache ccache, krb5_flags flags,
    const char *hostname, krb5_creds *in_creds, krb5_data *out_data);

krb5_error_code
krb5_fwd_tgt_creds(krb5_context context, krb5_auth_context auth_context,
    const char *hostname, krb5_principal client, krb5_principal server,
    krb5_ccache ccache, int forwardable, krb5_data *out_data);
```

DESCRIPTION

krb5_get_forwarded_creds() and **krb5_fwd_tgt_creds**() get tickets forwarded to *hostname*. If the tickets that are forwarded are address-less, the forwarded tickets will also be address-less, otherwise *hostname* will be used for figure out the address to forward the ticket too.

SEE ALSO

krb5(3), krb5_get_credentials(3), krb5.conf(5)

NAME

krb5_get_in_tkt, **krb5_get_in_cred**, **krb5_get_in_tkt_with_password**, **krb5_get_in_tkt_with_keytab**, **krb5_get_in_tkt_with_skey**, **krb5_free_kdc_rep**, **krb5_password_key_proc** — deprecated initial authentication functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_get_in_tkt(krb5_context context, krb5_flags options,
    const krb5_addresses *addrs, const krb5_enctype *etypes,
    const krb5_preauthtype *ptypes, krb5_key_proc key_proc,
    krb5_const_pointer keyseed, krb5_decrypt_proc decrypt_proc,
    krb5_const_pointer decryptarg, krb5_creds *creds, krb5_ccache ccache,
    krb5_kdc_rep *ret_as_reply);

krb5_error_code
krb5_get_in_cred(krb5_context context, krb5_flags options,
    const krb5_addresses *addrs, const krb5_enctype *etypes,
    const krb5_preauthtype *ptypes, const krb5_preauthdata *preauth,
    krb5_key_proc key_proc, krb5_const_pointer keyseed,
    krb5_decrypt_proc decrypt_proc, krb5_const_pointer decryptarg,
    krb5_creds *creds, krb5_kdc_rep *ret_as_reply);

krb5_error_code
krb5_get_in_tkt_with_password(krb5_context context, krb5_flags options,
    krb5_addresses *addrs, const krb5_enctype *etypes,
    const krb5_preauthtype *pre_auth_types, const char *password,
    krb5_ccache ccache, krb5_creds *creds, krb5_kdc_rep *ret_as_reply);

krb5_error_code
krb5_get_in_tkt_with_keytab(krb5_context context, krb5_flags options,
    krb5_addresses *addrs, const krb5_enctype *etypes,
    const krb5_preauthtype *pre_auth_types, krb5_keytab keytab,
    krb5_ccache ccache, krb5_creds *creds, krb5_kdc_rep *ret_as_reply);

krb5_error_code
krb5_get_in_tkt_with_skey(krb5_context context, krb5_flags options,
    krb5_addresses *addrs, const krb5_enctype *etypes,
    const krb5_preauthtype *pre_auth_types, const krb5_keyblock *key,
    krb5_ccache ccache, krb5_creds *creds, krb5_kdc_rep *ret_as_reply);

krb5_error_code
krb5_free_kdc_rep(krb5_context context, krb5_kdc_rep *rep);

krb5_error_code
krb5_password_key_proc(krb5_context context, krb5_enctype type,
    krb5_salt salt, krb5_const_pointer keyseed, krb5_keyblock **key);
```

DESCRIPTION

All the functions in this manual page are deprecated in the MIT implementation, and will soon be deprecated in Heimdal too, don't use them.

Getting initial credential ticket for a principal. **krb5_get_in_cred** is the function all other krb5_get_in function uses to fetch tickets. The other krb5_get_in function are more specialized and therefor somewhat easier to use.

If your need is only to verify a user and password, consider using krb5_verify_user(3) instead, it have a much simpler interface.

krb5_get_in_tkt and **krb5_get_in_cred** fetches initial credential, queries after key using the *key_proc* argument. The differences between the two function is that **krb5_get_in_tkt** stores the credential in a krb5_creds while **krb5_get_in_cred** stores the credential in a krb5_ccache.

krb5_get_in_tkt_with_password, **krb5_get_in_tkt_with_keytab**, and **krb5_get_in_tkt_with_skey** does the same work as **krb5_get_in_cred** but are more specialized.

krb5_get_in_tkt_with_password uses the clients password to authenticate. If the password argument is the user user queried with the default password query function.

krb5_get_in_tkt_with_keytab searches the given keytab for a service entry for the client principal. If the keytab is NULL the default keytab is used.

krb5_get_in_tkt_with_skey uses a key to get the initial credential.

There are some common arguments to the krb5_get_in functions, these are:

options are the KDC_OPT flags.

etypes is a NULL terminated array of encryption types that the client approves.

addrs a list of the addresses that the initial ticket. If it is NULL the list will be generated by the library.

pre_auth_types a NULL terminated array of pre-authentication types. If *pre_auth_types* is NULL the function will try without pre-authentication and return those pre-authentication that the KDC returned.

ret_as_reply will (if not NULL) be filled in with the response of the KDC and should be free with **krb5_free_kdc_rep**().

key_proc is a pointer to a function that should return a key salted appropriately. Using NULL will use the default password query function.

decrypt_proc Using NULL will use the default decryption function.

decryptarg will be passed to the decryption function *decrypt_proc*.

creds creds should be filled in with the template for a credential that should be requested. The client and server elements of the creds structure must be filled in. Upon return of the function it will be contain the content of the requested credential (*krb5_get_in_cred*), or it will be freed with krb5_free_creds(3) (all the other krb5_get_in functions).

ccache will store the credential in the credential cache *ccache*. The credential cache will not be initialized, thats up the the caller.

krb5_password_key_proc is a library function that is suitable using as the *krb5_key_proc* argument to **krb5_get_in_cred** or **krb5_get_in_tkt**. *keyseed* should be a pointer to a NUL terminated string or NULL. **krb5_password_key_proc** will query the user for the pass on the console if the password isn't given as the argument *keyseed*.

krb5_free_kdc_rep() frees the content of *rep*.

SEE ALSO

`krb5`(3), `krb5_verify_user`(3), `krb5.conf`(5), `kerberos`(8)

NAME

krb5_get_init_creds, **krb5_get_init_creds_keytab**, **krb5_get_init_creds_opt**, **krb5_get_init_creds_opt_alloc**, **krb5_get_init_creds_opt_free**, **krb5_get_init_creds_opt_init**, **krb5_get_init_creds_opt_set_address_list**, **krb5_get_init_creds_opt_set_addressless**, **krb5_get_init_creds_opt_set_anonymous**, **krb5_get_init_creds_opt_set_default_flags**, **krb5_get_init_creds_opt_set_etype_list**, **krb5_get_init_creds_opt_set_forwardable**, **krb5_get_init_creds_opt_set_pa_password**, **krb5_get_init_creds_opt_set_paq_request**, **krb5_get_init_creds_opt_set_preauth_list**, **krb5_get_init_creds_opt_set_proxiable**, **krb5_get_init_creds_opt_set_renew_life**, **krb5_get_init_creds_opt_set_salt**, **krb5_get_init_creds_opt_set_tkt_life**, **krb5_get_init_creds_opt_set_canonicalize**, **krb5_get_init_creds_opt_set_win2k**, **krb5_get_init_creds_password**, **krb5_prompt**, **krb5_prompter_posix** — Kerberos 5 initial authentication functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_get_init_creds_opt;

krb5_error_code
krb5_get_init_creds_opt_alloc(krb5_context context,
    krb5_get_init_creds_opt **opt);

void
krb5_get_init_creds_opt_free(krb5_context context,
    krb5_get_init_creds_opt *opt);

void
krb5_get_init_creds_opt_init(krb5_get_init_creds_opt *opt);

void
krb5_get_init_creds_opt_set_address_list(krb5_get_init_creds_opt *opt,
    krb5_addresses *addresses);

void
krb5_get_init_creds_opt_set_addressless(krb5_get_init_creds_opt *opt,
    krb5_boolean addressless);

void
krb5_get_init_creds_opt_set_anonymous(krb5_get_init_creds_opt *opt,
    int anonymous);

void
krb5_get_init_creds_opt_set_default_flags(krb5_context context,
    const char *appname, krb5_const_realm realm,
    krb5_get_init_creds_opt *opt);
```

void
krb5_get_init_creds_opt_set_etype_list(*krb5_get_init_creds_opt *opt*, *krb5_enctype *etype_list*, *int etype_list_length*);

void
krb5_get_init_creds_opt_set_forwardable(*krb5_get_init_creds_opt *opt*, *int forwardable*);

krb5_error_code
krb5_get_init_creds_opt_set_pa_password(*krb5_context context*, *krb5_get_init_creds_opt *opt*, *const char *password*, *krb5_s2k_proc key_proc*);

krb5_error_code
krb5_get_init_creds_opt_set_paq_request(*krb5_context context*, *krb5_get_init_creds_opt *opt*, *krb5_boolean req_pac*);

krb5_error_code
krb5_get_init_creds_opt_set_pkinit(*krb5_context context*, *krb5_get_init_creds_opt *opt*, *const char *cert_file*, *const char *key_file*, *const char *x509_anchors*, *int flags*, *char *password*);

void
krb5_get_init_creds_opt_set_preauth_list(*krb5_get_init_creds_opt *opt*, *krb5_preauthtype *preauth_list*, *int preauth_list_length*);

void
krb5_get_init_creds_opt_set_proxiable(*krb5_get_init_creds_opt *opt*, *int proxiable*);

void
krb5_get_init_creds_opt_set_renew_life(*krb5_get_init_creds_opt *opt*, *krb5_deltat renew_life*);

void
krb5_get_init_creds_opt_set_salt(*krb5_get_init_creds_opt *opt*, *krb5_data *salt*);

void
krb5_get_init_creds_opt_set_tkt_life(*krb5_get_init_creds_opt *opt*, *krb5_deltat tkt_life*);

krb5_error_code
krb5_get_init_creds_opt_set_canonicalize(*krb5_context context*, *krb5_get_init_creds_opt *opt*, *krb5_boolean req*);

krb5_error_code
krb5_get_init_creds_opt_set_win2k(*krb5_context context*, *krb5_get_init_creds_opt *opt*, *krb5_boolean req*);

krb5_error_code
krb5_get_init_creds(*krb5_context context*, *krb5_creds *creds*, *krb5_principal client*, *krb5_prompter_fct prompter*, *void *prompter_data*, *krb5_deltat start_time*, *const char *in_tkt_service*, *krb5_get_init_creds_opt *options*);

krb5_error_code
krb5_get_init_creds_password(*krb5_context context*, *krb5_creds *creds*,
krb5_principal client, *const char *password*,
krb5_prompter_fct prompter, *void *prompter_data*,
krb5_deltat start_time, *const char *in_tkt_service*,
*krb5_get_init_creds_opt *in_options*);

krb5_error_code
krb5_get_init_creds_keytab(*krb5_context context*, *krb5_creds *creds*,
krb5_principal client, *krb5_keytab keytab*, *krb5_deltat start_time*,
*const char *in_tkt_service*, *krb5_get_init_creds_opt *options*);

int
krb5_prompter_posix(*krb5_context context*, *void *data*, *const char *name*,
*const char *banner*, *int num_prompts*, *krb5_prompt prompts[]*);

DESCRIPTION

Getting initial credential ticket for a principal. That may include changing an expired password, and doing preauthentication. This interface that replaces the deprecated *krb5_in_tkt* and *krb5_in_cred* functions.

If you only want to verify a username and password, consider using krb5_verify_user(3) instead, since it also verifies that initial credentials with using a keytab to make sure the response was from the KDC.

First a krb5_get_init_creds_opt structure is initialized with **krb5_get_init_creds_opt_alloc**() or **krb5_get_init_creds_opt_init**(). **krb5_get_init_creds_opt_alloc**() allocates a extendible structures that needs to be freed with **krb5_get_init_creds_opt_free**(). The structure may be modified by any of the **krb5_get_init_creds_opt_set**() functions to change request parameters and authentication information.

If the caller want to use the default options, NULL can be passed instead.

The the actual request to the KDC is done by any of the **krb5_get_init_creds**(), **krb5_get_init_creds_password**(), or **krb5_get_init_creds_keytab**() functions. **krb5_get_init_creds**() is the least specialized function and can, with the right in data, behave like the latter two. The latter two are there for compatibility with older releases and they are slightly easier to use.

krb5_prompt is a structure containing the following elements:

```
typedef struct {
    const char *prompt;
    int hidden;
    krb5_data *reply;
    krb5_prompt_type type
} krb5_prompt;
```

prompt is the prompt that should shown to the user If *hidden* is set, the prompter function shouldn't echo the output to the display device. *reply* must be preallocated; it will not be allocated by the prompter function. Possible values for the *type* element are:

KRB5_PROMPT_TYPE_PASSWORD
KRB5_PROMPT_TYPE_NEW_PASSWORD
KRB5_PROMPT_TYPE_NEW_PASSWORD_AGAIN

KRB5_PROMPT_TYPE_PREAUTH
KRB5_PROMPT_TYPE_INFO

krb5_prompter_posix() is the default prompter function in a POSIX environment. It matches the *krb5_prompter_fct* and can be used in the *krb5_get_init_creds* functions. **krb5_prompter_posix**() doesn't require *prompter_data.*

If the *start_time* is zero, then the requested ticket will be valid beginning immediately. Otherwise, the *start_time* indicates how far in the future the ticket should be postdated.

If the *in_tkt_service* name is non-NULL, that principal name will be used as the server name for the initial ticket request. The realm of the name specified will be ignored and will be set to the realm of the client name. If no in_tkt_service name is specified, krbtgt/CLIENT-REALM@CLIENT-REALM will be used.

For the rest of arguments, a configuration or library default will be used if no value is specified in the options structure.

krb5_get_init_creds_opt_set_address_list() sets the list of *addresses* that is should be stored in the ticket.

krb5_get_init_creds_opt_set_addressless() controls if the ticket is requested with addresses or not, **krb5_get_init_creds_opt_set_address_list**() overrides this option.

krb5_get_init_creds_opt_set_anonymous() make the request anonymous if the *anonymous* parameter is non-zero.

krb5_get_init_creds_opt_set_default_flags() sets the default flags using the configuration file.

krb5_get_init_creds_opt_set_etype_list() set a list of enctypes that the client is willing to support in the request.

krb5_get_init_creds_opt_set_forwardable() request a forwardable ticket.

krb5_get_init_creds_opt_set_pa_password() set the *password* and *key_proc* that is going to be used to get a new ticket. *password* or *key_proc* can be NULL if the caller wants to use the default values. If the *password* is unset and needed, the user will be prompted for it.

krb5_get_init_creds_opt_set_paq_request() sets the password that is going to be used to get a new ticket.

krb5_get_init_creds_opt_set_preauth_list() sets the list of client-supported preauth types.

krb5_get_init_creds_opt_set_proxiable() makes the request proxiable.

krb5_get_init_creds_opt_set_renew_life() sets the requested renewable lifetime.

krb5_get_init_creds_opt_set_salt() sets the salt that is going to be used in the request.

krb5_get_init_creds_opt_set_tkt_life() sets requested ticket lifetime.

krb5_get_init_creds_opt_set_canonicalize() requests that the KDC canonicalize the client pricipal if possible.

krb5_get_init_creds_opt_set_win2k() turns on compatibility with Windows 2000.

SEE ALSO

krb5(3), krb5_creds(3), krb5_verify_user(3), krb5.conf(5), kerberos(8)

NAME

krb5_get_krbhst, **krb5_get_krb_admin_hst**, **krb5_get_krb_changepw_hst**, **krb5_get_krb524hst**, **krb5_free_krbhst** — lookup Kerberos KDC hosts

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_get_krbhst(krb5_context context, const krb5_realm *realm,
    char ***hostlist);

krb5_error_code
krb5_get_krb_admin_hst(krb5_context context, const krb5_realm *realm,
    char ***hostlist);

krb5_error_code
krb5_get_krb_changepw_hst(krb5_context context, const krb5_realm *realm,
    char ***hostlist);

krb5_error_code
krb5_get_krb524hst(krb5_context context, const krb5_realm *realm,
    char ***hostlist);

krb5_error_code
krb5_free_krbhst(krb5_context context, char **hostlist);
```

DESCRIPTION

These functions implement the old API to get a list of Kerberos hosts, and are thus similar to the **krb5_krbhst_init**() functions. However, since these functions returns *all* hosts in one go, they potentially have to do more lookups than necessary. These functions remain for compatibility reasons.

After a call to one of these functions, *hostlist* is a NULL terminated list of strings, pointing to the requested Kerberos hosts. These should be freed with **krb5_free_krbhst**() when done with.

EXAMPLES

The following code will print the KDCs of the realm “MY.REALM”.

```
char **hosts, **p;
krb5_get_krbhst(context, "MY.REALM", &hosts);
for(p = hosts; *p; p++)
    printf("%s\n", *p);
krb5_free_krbhst(context, hosts);
```

SEE ALSO

krb5_krbhst_init(3)

NAME

krb5_getportbyname — get port number by name

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

int
krb5_getportbyname(krb5_context context, const char *service,
    const char *proto, int default_port);
```

DESCRIPTION

krb5_getportbyname() gets the port number for *service / proto* pair from the global service table for and returns it in network order. If it isn't found in the global table, the *default_port* (given in host order) is returned.

EXAMPLE

```
int port = krb5_getportbyname(context, "kerberos", "tcp", 88);
```

SEE ALSO

krb5(3)

NAME

krb5_add_et_list, **krb5_add_extra_addresses**, **krb5_add_ignore_addresses**, **krb5_context**, **krb5_free_config_files**, **krb5_free_context**, **krb5_get_default_config_files**, **krb5_get_dns_canonize_hostname**, **krb5_get_extra_addresses**, **krb5_get_fcache_version**, **krb5_get_ignore_addresses**, **krb5_get_kdc_sec_offset**, **krb5_get_max_time_skew**, **krb5_get_use_admin_kdc** **krb5_init_context**, **krb5_init_ets**, **krb5_prepend_config_files**, **krb5_prepend_config_files_default**, **krb5_set_config_files**, **krb5_set_dns_canonize_hostname**, **krb5_set_extra_addresses**, **krb5_set_fcache_version**, **krb5_set_ignore_addresses**, **krb5_set_max_time_skew**, **krb5_set_use_admin_kdc**, — create, modify and delete krb5_context structures

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

struct krb5_context;

krb5_error_code
krb5_init_context(krb5_context *context);

void
krb5_free_context(krb5_context context);

void
krb5_init_ets(krb5_context context);

krb5_error_code
krb5_add_et_list(krb5_context context,void (*func)(struct et_list **));

krb5_error_code
krb5_add_extra_addresses(krb5_context context,krb5_addresses *addresses);

krb5_error_code
krb5_set_extra_addresses(krb5_context context,
    const krb5_addresses *addresses);

krb5_error_code
krb5_get_extra_addresses(krb5_context context,krb5_addresses *addresses);

krb5_error_code
krb5_add_ignore_addresses(krb5_context context,krb5_addresses *addresses);

krb5_error_code
krb5_set_ignore_addresses(krb5_context context,
    const krb5_addresses *addresses);

krb5_error_code
krb5_get_ignore_addresses(krb5_context context,krb5_addresses *addresses);

krb5_error_code
krb5_set_fcache_version(krb5_context context,int version);

krb5_error_code
krb5_get_fcache_version(krb5_context context,int *version);
```

void
krb5_set_dns_canonize_hostname(*krb5_context context*, *krb5_boolean flag*);

krb5_boolean
krb5_get_dns_canonize_hostname(*krb5_context context*);

krb5_error_code
krb5_get_kdc_sec_offset(*krb5_context context*, *int32_t *sec*, *int32_t *usec*);

krb5_error_code
krb5_set_config_files(*krb5_context context*, *char **filenames*);

krb5_error_code
krb5_prepend_config_files(*const char *filelist*, *char **pq*, *char ***ret_pp*);

krb5_error_code
krb5_prepend_config_files_default(*const char *filelist*,
*char ***pfilenames*);

krb5_error_code
krb5_get_default_config_files(*char ***pfilenames*);

void
krb5_free_config_files(*char **filenames*);

void
krb5_set_use_admin_kdc(*krb5_context context*, *krb5_boolean flag*);

krb5_boolean
krb5_get_use_admin_kdc(*krb5_context context*);

time_t
krb5_get_max_time_skew(*krb5_context context*);

krb5_error_code
krb5_set_max_time_skew(*krb5_context context*, *time_t time*);

DESCRIPTION

The **krb5_init_context**() function initializes the *context* structure and reads the configuration file `/etc/krb5.conf`.

The structure should be freed by calling **krb5_free_context**() when it is no longer being used.

krb5_init_context() returns 0 to indicate success. Otherwise an errno code is returned. Failure means either that something bad happened during initialization (typically [ENOMEM]) or that Kerberos should not be used [ENXIO].

krb5_init_ets() adds all `com_err`(3) libs to *context*. This is done by **krb5_init_context**().

krb5_add_et_list() adds a `com_err`(3) error-code handler *func* to the specified *context*. The error handler must generated by the the re-rentrant version of the `compile_et`(3) program. **krb5_add_extra_addresses**() add a list of addresses that should be added when requesting tickets.

krb5_add_ignore_addresses() add a list of addresses that should be ignored when requesting tickets.

krb5_get_extra_addresses() get the list of addresses that should be added when requesting tickets.

krb5_get_ignore_addresses() get the list of addresses that should be ignored when requesting tickets.

krb5_set_ignore_addresses() set the list of addresses that should be ignored when requesting tickets.

krb5_set_extra_addresses() set the list of addresses that should be added when requesting tickets.

krb5_set_fcache_version() sets the version of file credentials caches that should be used.

krb5_get_fcache_version() gets the version of file credentials caches that should be used.

krb5_set_dns_canonize_hostname() sets if the context is configured to canonicalize hostnames using DNS.

krb5_get_dns_canonize_hostname() returns if the context is configured to canonicalize hostnames using DNS.

krb5_get_kdc_sec_offset() returns the offset between the localtime and the KDC's time. *sec* and *usec* are both optional argument and NULL can be passed in.

krb5_set_config_files() set the list of configuration files to use and re-initialize the configuration from the files.

krb5_prepend_config_files() parse the *filelist* and prepend the result to the already existing list *pq* The result is returned in *ret_pp* and should be freed with **krb5_free_config_files**().

krb5_prepend_config_files_default() parse the *filelist* and append that to the default list of configuration files.

krb5_get_default_config_files() get a list of default configuration files.

krb5_free_config_files() free a list of configuration files returned by **krb5_get_default_config_files**(), **krb5_prepend_config_files_default**(), or **krb5_prepend_config_files**().

krb5_set_use_admin_kdc() sets if all KDC requests should go admin KDC.

krb5_get_use_admin_kdc() gets if all KDC requests should go admin KDC.

krb5_get_max_time_skew() and **krb5_set_max_time_skew**() get and sets the maximum allowed time skew between client and server.

SEE ALSO

errno(2), krb5(3), krb5_config(3), krb5_context(3), kerberos(8)

NAME

krb5_is_thread_safe — is the Kerberos library compiled with multithread support

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_boolean
krb5_is_thread_safe(void);
```

DESCRIPTION

krb5_is_thread_safe returns TRUE if the library was compiled with with multithread support. If the library isn't compiled, the consumer have to use a global lock to make sure Kerboros functions are not called at the same time by diffrent threads.

SEE ALSO

krb5_create_checksum(3), krb5_encrypt(3)

NAME

krb5_keyblock, **krb5_keyblock_get_enctype**, **krb5_copy_keyblock**, **krb5_copy_keyblock_contents**, **krb5_free_keyblock**, **krb5_free_keyblock_contents**, **krb5_generate_random_keyblock**, **krb5_generate_subkey**, **krb5_generate_subkey_extended**, **krb5_keyblock_init**, **krb5_keyblock_zero**, **krb5_random_to_key** — Kerberos 5 key handling functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_keyblock;

krb5_enctype
krb5_keyblock_get_enctype(const krb5_keyblock *block);

krb5_error_code
krb5_copy_keyblock(krb5_context context, krb5_keyblock **to);

krb5_error_code
krb5_copy_keyblock_contents(krb5_context context,
    const krb5_keyblock *inblock, krb5_keyblock *to);

void
krb5_free_keyblock(krb5_context context, krb5_keyblock *keyblock);

void
krb5_free_keyblock_contents(krb5_context context, krb5_keyblock *keyblock);

krb5_error_code
krb5_generate_random_keyblock(krb5_context context, krb5_enctype type,
    krb5_keyblock *key);

krb5_error_code
krb5_generate_subkey(krb5_context context, const krb5_keyblock *key,
    krb5_keyblock **subkey);

krb5_error_code
krb5_generate_subkey_extended(krb5_context context,
    const krb5_keyblock *key, krb5_enctype enctype,
    krb5_keyblock **subkey);

krb5_error_code
krb5_keyblock_init(krb5_context context, krb5_enctype type,
    const void *data, size_t size, krb5_keyblock *key);

void
krb5_keyblock_zero(krb5_keyblock *keyblock);

krb5_error_code
krb5_random_to_key(krb5_context context, krb5_enctype type,
    const void *data, size_t size, krb5_keyblock *key);
```

DESCRIPTION

`krb5_keyblock` holds the encryption key for a specific encryption type. There is no component inside `krb5_keyblock` that is directly referable.

krb5_keyblock_get_enctype() returns the encryption type of the keyblock.

krb5_copy_keyblock() makes a copy the keyblock *inblock* to the output *out*. *out* should be freed by the caller with *krb5_free_keyblock*.

krb5_copy_keyblock_contents() copies the contents of *inblock* to the *to* keyblock. The destination keyblock is overritten.

krb5_free_keyblock() zeros out and frees the content and the keyblock itself.

krb5_free_keyblock_contents() zeros out and frees the content of the keyblock.

krb5_generate_random_keyblock() creates a new content of the keyblock *key* of type encrytion type *type*. The content of *key* is overwritten and not freed, so the caller should be sure it is freed before calling the function.

krb5_generate_subkey() generates a *subkey* of the same type as *key*. The caller must free the subkey with *krb5_free_keyblock*.

krb5_generate_subkey_extended() generates a *subkey* of the specified encryption type *type*. If *type* is `ETYPE_NULL`, of the same type as *key*. The caller must free the subkey with *krb5_free_keyblock*.

krb5_keyblock_init() Fill in *key* with key data of type *enctype* from *data* of length *size*. Key should be freed using **krb5_free_keyblock_contents**().

krb5_keyblock_zero() zeros out the keyblock to to make sure no keymaterial is in memory. Note that **krb5_free_keyblock_contents**() also zeros out the memory.

krb5_random_to_key() converts the random bytestring to a protocol key according to Kerberos crypto frame work. It the resulting key will be of type *enctype*. It may be assumed that all the bits of the input string are equally random, even though the entropy present in the random source may be limited

SEE ALSO

`krb5_crypto_init`(3), `krb5`(3), `krb5.conf`(5)

NAME

krb5_kt_ops, **krb5_keytab_entry**, **krb5_kt_cursor**, **krb5_kt_add_entry**, **krb5_kt_close**, **krb5_kt_compare**, **krb5_kt_copy_entry_contents**, **krb5_kt_default**, **krb5_kt_default_modify_name**, **krb5_kt_default_name**, **krb5_kt_end_seq_get**, **krb5_kt_free_entry**, **krb5_kt_get_entry**, **krb5_kt_get_name**, **krb5_kt_get_type**, **krb5_kt_next_entry**, **krb5_kt_read_service_key**, **krb5_kt_register**, **krb5_kt_remove_entry**, **krb5_kt_resolve**, **krb5_kt_start_seq_get** — manage keytab (key storage) files

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_kt_add_entry(krb5_context context, krb5_keytab id,
    krb5_keytab_entry *entry);

krb5_error_code
krb5_kt_close(krb5_context context, krb5_keytab id);

krb5_boolean
krb5_kt_compare(krb5_context context, krb5_keytab_entry *entry,
    krb5_const_principal principal, krb5_kvno vno, krb5_enctype enctype);

krb5_error_code
krb5_kt_copy_entry_contents(krb5_context context,
    const krb5_keytab_entry *in, krb5_keytab_entry *out);

krb5_error_code
krb5_kt_default(krb5_context context, krb5_keytab *id);

krb5_error_code
krb5_kt_default_modify_name(krb5_context context, char *name,
    size_t namesize);

krb5_error_code
krb5_kt_default_name(krb5_context context, char *name, size_t namesize);

krb5_error_code
krb5_kt_end_seq_get(krb5_context context, krb5_keytab id,
    krb5_kt_cursor *cursor);

krb5_error_code
krb5_kt_free_entry(krb5_context context, krb5_keytab_entry *entry);

krb5_error_code
krb5_kt_get_entry(krb5_context context, krb5_keytab id,
    krb5_const_principal principal, krb5_kvno kvno, krb5_enctype enctype,
    krb5_keytab_entry *entry);

krb5_error_code
krb5_kt_get_name(krb5_context context, krb5_keytab keytab, char *name,
    size_t namesize);
```

krb5_error_code
krb5_kt_get_type(*krb5_context context*, *krb5_keytab keytab*, *char *prefix*, *size_t prefixsize*);

krb5_error_code
krb5_kt_next_entry(*krb5_context context*, *krb5_keytab id*, *krb5_keytab_entry *entry*, *krb5_kt_cursor *cursor*);

krb5_error_code
krb5_kt_read_service_key(*krb5_context context*, *krb5_pointer keyprocarg*, *krb5_principal principal*, *krb5_kvno vno*, *krb5_enctype enctype*, *krb5_keyblock **key*);

krb5_error_code
krb5_kt_register(*krb5_context context*, *const krb5_kt_ops *ops*);

krb5_error_code
krb5_kt_remove_entry(*krb5_context context*, *krb5_keytab id*, *krb5_keytab_entry *entry*);

krb5_error_code
krb5_kt_resolve(*krb5_context context*, *const char *name*, *krb5_keytab *id*);

krb5_error_code
krb5_kt_start_seq_get(*krb5_context context*, *krb5_keytab id*, *krb5_kt_cursor *cursor*);

DESCRIPTION

A keytab name is on the form `type:residual`. The `residual` part is specific to each keytab-type.

When a keytab-name is resolved, the type is matched with an internal list of keytab types. If there is no matching keytab type, the default keytab is used. The current default type is **`file`**. The default value can be changed in the configuration file `/etc/krb5.conf` by setting the variable `[defaults]default_keytab_name`.

The keytab types that are implemented in Heimdal are:

`file` store the keytab in a file, the type's name is `FILE`. The residual part is a filename. For compatibility with other Kerberos implemtation `WRFILE` and is also accepted. `WRFILE` has the same format as `FILE`. `JAVA14` have a format that is compatible with older versions of MIT kerberos and SUN's Java based installation. They store a truncted kvno, so when the knvo excess 255, they are truncted in this format.

`keyfile`
store the keytab in a `AFS` keyfile (usually `/usr/afs/etc/KeyFile`), the type's name is `AFSKEYFILE`. The residual part is a filename.

`krb4` the keytab is a Kerberos 4 `srvtab` that is on-the-fly converted to a keytab. The type's name is `krb4`. The residual part is a filename.

`memory`
The keytab is stored in a memory segment. This allows sensitive and/or temporary data not to be stored on disk. The type's name is `MEMORY`. Each `MEMORY` keytab is referenced counted by and opened by the residual name, so two handles can point to the same memory area. When the last user closes the entry, it disappears.

`krb5_keytab_entry` holds all data for an entry in a keytab file, like principal name, key-type, key, key-version number, etc. **`krb5_kt_cursor`** holds the current position that is used when iterating through a

keytab entry with **krb5_kt_start_seq_get**(), **krb5_kt_next_entry**(), and **krb5_kt_end_seq_get**().

krb5_kt_ops contains the different operations that can be done to a keytab. This structure is normally only used when doing a new keytab-type implementation.

krb5_kt_resolve() is the equivalent of an open(2) on keytab. Resolve the keytab name in *name* into a keytab in *id*. Returns 0 or an error. The opposite of **krb5_kt_resolve**() is **krb5_kt_close**().

krb5_kt_close() frees all resources allocated to the keytab, even on failure. Returns 0 or an error.

krb5_kt_default() sets the argument *id* to the default keytab. Returns 0 or an error.

krb5_kt_default_modify_name() copies the name of the default modify keytab into *name*. Return 0 or KRB5_CONFIG_NOTENUFSPACE if *namesize* is too short.

krb5_kt_default_name() copies the name of the default keytab into *name*. Return 0 or KRB5_CONFIG_NOTENUFSPACE if *namesize* is too short.

krb5_kt_add_entry() adds a new *entry* to the keytab *id*. KRB5_KT_NOWRITE is returned if the keytab is a readonly keytab.

krb5_kt_compare() compares the passed in *entry* against *principal*, *vno*, and *enctype*. Any of *principal*, *vno* or *enctype* might be 0 which acts as a wildcard. Return TRUE if they compare the same, FALSE otherwise.

krb5_kt_copy_entry_contents() copies the contents of *in* into *out*. Returns 0 or an error.

krb5_kt_get_name() retrieves the name of the keytab *keytab* into *name*, *namesize*. Returns 0 or an error.

krb5_kt_get_type() retrieves the type of the keytab *keytab* and store the prefix/name for type of the keytab into *prefix*, *prefixsize*. The prefix will have the maximum length of KRB5_KT_PREFIX_MAX_LEN (including terminating NUL). Returns 0 or an error.

krb5_kt_free_entry() frees the contents of *entry*.

krb5_kt_start_seq_get() sets *cursor* to point at the beginning of *id*. Returns 0 or an error.

krb5_kt_next_entry() gets the next entry from *id* pointed to by *cursor* and advance the *cursor*. On success the returne entry must be freed with **krb5_kt_free_entry**(). Returns 0 or an error.

krb5_kt_end_seq_get() releases all resources associated with *cursor*.

krb5_kt_get_entry() retrieves the keytab entry for *principal*, *kvno*, *enctype* into *entry* from the keytab *id*. When comparing an entry in the keytab to determine a match, the function **krb5_kt_compare**() is used, so the wildcard rules applies to the argument of too. On success the returne entry must be freed with **krb5_kt_free_entry**(). Returns 0 or an error.

krb5_kt_read_service_key() reads the key identified by (*principal*, *vno*, *enctype*) from the keytab in *keyprocarg* (the system default keytab if NULL is used) into ∗*key*. *keyprocarg* is the same argument as to *name* argument to **krb5_kt_resolve**(). Internal **krb5_kt_compare**() will be used, so the same wildcard rules applies to **krb5_kt_read_service_key**(). On success the returned key must be freed with *krb5_free_keyblock*. Returns 0 or an error.

krb5_kt_remove_entry() removes the entry *entry* from the keytab *id*. When comparing an entry in the keytab to determine a match, the function **krb5_kt_compare**() is use, so the wildcard rules applies to the argument of **krb5_kt_remove_entry**(). Returns 0, KRB5_KT_NOTFOUND if not entry matched or another error.

krb5_kt_register() registers a new keytab type *ops*. Returns 0 or an error.

EXAMPLES

This is a minimalistic version of **ktutil**.

```
int
main (int argc, char **argv)
{
    krb5_context context;
    krb5_keytab keytab;
    krb5_kt_cursor cursor;
    krb5_keytab_entry entry;
    krb5_error_code ret;
    char *principal;

    if (krb5_init_context (&context) != 0)
        errx(1, "krb5_context");

    ret = krb5_kt_default (context, &keytab);
    if (ret)
        krb5_err(context, 1, ret, "krb5_kt_default");

    ret = krb5_kt_start_seq_get(context, keytab, &cursor);
    if (ret)
        krb5_err(context, 1, ret, "krb5_kt_start_seq_get");
    while((ret = krb5_kt_next_entry(context, keytab, &entry, &cursor)) == 0){
        krb5_unparse_name_short(context, entry.principal, &principal);
        printf("principal: %s\n", principal);
        free(principal);
        krb5_kt_free_entry(context, &entry);
    }
    ret = krb5_kt_end_seq_get(context, keytab, &cursor);
    if (ret)
        krb5_err(context, 1, ret, "krb5_kt_end_seq_get");
    ret = krb5_kt_close(context, keytab);
    if (ret)
        krb5_err(context, 1, ret, "krb5_kt_close");
    krb5_free_context(context);
    return 0;
}
```

COMPATIBILITY

Heimdal stored the ticket flags in machine bit-field order before Heimdal 0.7. The behavior is possible to change in with the option `[libdefaults]fcc-mit-ticketflags`. Heimdal 0.7 also code to detech that ticket flags was in the wrong order and correct them. This matters when doing delegation in GSS-API because the client code looks at the flag to determin if it is possible to do delegation if the user requested it.

SEE ALSO

krb5.conf(5), kerberos(8)

NAME

krb5_krbhst_init, **krb5_krbhst_init_flags**, **krb5_krbhst_next**, **krb5_krbhst_next_as_string**, **krb5_krbhst_reset**, **krb5_krbhst_free**, **krb5_krbhst_format_string**, **krb5_krbhst_get_addrinfo** — lookup Kerberos KDC hosts

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_krbhst_init(krb5_context context, const char *realm,
    unsigned int type, krb5_krbhst_handle *handle);

krb5_error_code
krb5_krbhst_init_flags(krb5_context context, const char *realm,
    unsigned int type, int flags, krb5_krbhst_handle *handle);

krb5_error_code
krb5_krbhst_next(krb5_context context, krb5_krbhst_handle handle,
    krb5_krbhst_info **host);

krb5_error_code
krb5_krbhst_next_as_string(krb5_context context,
    krb5_krbhst_handle handle, char *hostname, size_t hostlen);

void
krb5_krbhst_reset(krb5_context context, krb5_krbhst_handle handle);

void
krb5_krbhst_free(krb5_context context, krb5_krbhst_handle handle);

krb5_error_code
krb5_krbhst_format_string(krb5_context context,
    const krb5_krbhst_info *host, char *hostname, size_t hostlen);

krb5_error_code
krb5_krbhst_get_addrinfo(krb5_context context, krb5_krbhst_info *host,
    struct addrinfo **ai);
```

DESCRIPTION

These functions are used to sequence through all Kerberos hosts of a particular realm and service. The service type can be the KDCs, the administrative servers, the password changing servers, or the servers for Kerberos 4 ticket conversion.

First a handle to a particular service is obtained by calling **krb5_krbhst_init**() (or **krb5_krbhst_init_flags**()) with the *realm* of interest and the type of service to lookup. The *type* can be one of:

KRB5_KRBHST_KDC
KRB5_KRBHST_ADMIN
KRB5_KRBHST_CHANGEPW
KRB5_KRBHST_KRB524

The *handle* is returned to the caller, and should be passed to the other functions.

The *flag* argument to **krb5_krbhst_init_flags** is the same flags as **krb5_send_to_kdc_flags**() uses. Possible values are:

KRB5_KRBHST_FLAGS_MASTER only talk to master (readwrite) KDC
KRB5_KRBHST_FLAGS_LARGE_MSG
this is a large message, so use transport that can handle that.

For each call to **krb5_krbhst_next**() information on a new host is returned. The former function returns in *host* a pointer to a structure containing information about the host, such as protocol, hostname, and port:

```
typedef struct krb5_krbhst_info {
    enum { KRB5_KRBHST_UDP,
           KRB5_KRBHST_TCP,
           KRB5_KRBHST_HTTP } proto;
    unsigned short port;
    struct addrinfo *ai;
    struct krb5_krbhst_info *next;
    char hostname[1];
} krb5_krbhst_info;
```

The related function, **krb5_krbhst_next_as_string**(), return the same information as a URL-like string.

When there are no more hosts, these functions return KRB5_KDC_UNREACH.

To re-iterate over all hosts, call **krb5_krbhst_reset**() and the next call to **krb5_krbhst_next**() will return the first host.

When done with the handle, **krb5_krbhst_free**() should be called.

To use a *krb5_krbhst_info*, there are two functions: **krb5_krbhst_format_string**() that will return a printable representation of that struct and **krb5_krbhst_get_addrinfo**() that will return a *struct addrinfo* that can then be used for communicating with the server mentioned.

EXAMPLES

The following code will print the KDCs of the realm “MY.REALM”:

```
krb5_krbhst_handle handle;
char host[MAXHOSTNAMELEN];
krb5_krbhst_init(context, "MY.REALM", KRB5_KRBHST_KDC, &handle);
while(krb5_krbhst_next_as_string(context, handle,
                                 host, sizeof(host)) == 0)
    printf("%s\n", host);
krb5_krbhst_free(context, handle);
```

SEE ALSO

getaddrinfo(3), krb5_get_krbhst(3), krb5_send_to_kdc_flags(3)

HISTORY

These functions first appeared in Heimdal 0.3g.

NAME

krb5_kuserok — checks if a principal is permitted to login as a user

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_boolean
krb5_kuserok(krb5_context context, krb5_principal principal,
    const char *user);
```

DESCRIPTION

This function takes the name of a local *user* and checks if *principal* is allowed to log in as that user.

The *user* may have a `~/.k5login` file listing principals that are allowed to login as that user. If that file does not exist, all principals with a first component identical to the username, and a realm considered local, are allowed access.

The `.k5login` file must contain one principal per line, be owned by *user*, and not be writable by group or other (but must be readable by anyone).

Note that if the file exists, no implicit access rights are given to *user*@⟨localrealm⟩.

Optionally, a set of files may be put in `~/.k5login.d` (a directory), in which case they will all be checked in the same manner as `.k5login`. The files may be called anything, but files starting with a hash ("#"), or ending with a tilde ("~") are ignored. Subdirectories are not traversed. Note that this directory may not be checked by other implementations.

RETURN VALUES

krb5_kuserok returns `TRUE` if access should be granted, `FALSE` otherwise.

HISTORY

The `~/.k5login.d` feature appeared in Heimdal 0.7.

SEE ALSO

`krb5_get_default_realms`(3), `krb5_verify_user`(3), `krb5_verify_user_lrealm`(3), `krb5_verify_user_opt`(3), `krb5.conf`(5)

NAME

krb5_mk_req, **krb5_mk_req_exact**, **krb5_mk_req_extended**, **krb5_rd_req**, **krb5_rd_req_with_keyblock**, **krb5_mk_rep**, **krb5_mk_rep_exact**, **krb5_mk_rep_extended**, **krb5_rd_rep**, **krb5_build_ap_req**, **krb5_verify_ap_req** — create and read application authentication request

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_mk_req(krb5_context context, krb5_auth_context *auth_context,
    const krb5_flags ap_req_options, const char *service,
    const char *hostname, krb5_data *in_data, krb5_ccache ccache,
    krb5_data *outbuf);

krb5_error_code
krb5_mk_req_extended(krb5_context context,
    krb5_auth_context *auth_context, const krb5_flags ap_req_options,
    krb5_data *in_data, krb5_creds *in_creds, krb5_data *outbuf);

krb5_error_code
krb5_rd_req(krb5_context context, krb5_auth_context *auth_context,
    const krb5_data *inbuf, krb5_const_principal server,
    krb5_keytab keytab, krb5_flags *ap_req_options, krb5_ticket **ticket);

krb5_error_code
krb5_build_ap_req(krb5_context context, krb5_enctype enctype,
    krb5_creds *cred, krb5_flags ap_options, krb5_data authenticator,
    krb5_data *retdata);

krb5_error_code
krb5_verify_ap_req(krb5_context context, krb5_auth_context *auth_context,
    krb5_ap_req *ap_req, krb5_const_principal server,
    krb5_keyblock *keyblock, krb5_flags flags, krb5_flags *ap_req_options,
    krb5_ticket **ticket);
```

DESCRIPTION

The functions documented in this manual page document the functions that facilitates the exchange between a Kerberos client and server. They are the core functions used in the authentication exchange between the client and the server.

The **krb5_mk_req** and **krb5_mk_req_extended** creates the Kerberos message KRB_AP_REQ that is sent from the client to the server as the first packet in a client/server exchange. The result that should be sent to server is stored in *outbuf*.

auth_context should be allocated with **krb5_auth_con_init**() or NULL passed in, in that case, it will be allocated and freed internally.

The input data *in_data* will have a checksum calculated over it and checksum will be transported in the message to the server.

ap_req_options can be set to one or more of the following flags:

`AP_OPTS_USE_SESSION_KEY`
: Use the session key when creating the request, used for user to user authentication.

`AP_OPTS_MUTUAL_REQUIRED`
: Mark the request as mutual authenticate required so that the receiver returns a mutual authentication packet.

The **krb5_rd_req** read the AP_REQ in *inbuf* and verify and extract the content. If *server* is specified, that server will be fetched from the *keytab* and used unconditionally. If *server* is `NULL`, the *keytab* will be search for a matching principal.

The *keytab* argument specifies what keytab to search for receiving principals. The arguments *ap_req_options* and *ticket* returns the content.

When the AS-REQ is a user to user request, neither of *keytab* or *principal* are used, instead **krb5_rd_req**() expects the session key to be set in *auth_context*.

The **krb5_verify_ap_req** and **krb5_build_ap_req** both constructs and verify the AP_REQ message, should not be used by external code.

SEE ALSO

`krb5`(3), `krb5.conf`(5)

NAME

krb5_mk_safe, **krb5_mk_priv** — generates integrity protected and/or encrypted messages

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_mk_priv(krb5_context context, krb5_auth_context auth_context,
    const krb5_data *userdata, krb5_data *outbuf,
    krb5_replay_data *outdata);

krb5_error_code
krb5_mk_safe(krb5_context context, krb5_auth_context auth_context,
    const krb5_data *userdata, krb5_data *outbuf,
    krb5_replay_data *outdata);
```

DESCRIPTION

krb5_mk_safe() and **krb5_mk_priv**() formats KRB-SAFE (integrity protected) and KRB-PRIV (also encrypted) messages into *outbuf*. The actual message data is taken from *userdata*. If the KRB5_AUTH_CONTEXT_DO_SEQUENCE or KRB5_AUTH_CONTEXT_DO_TIME flags are set in the *auth_context*, sequence numbers and time stamps are generated. If the KRB5_AUTH_CONTEXT_RET_SEQUENCE or KRB5_AUTH_CONTEXT_RET_TIME flags are set they are also returned in the *outdata* parameter.

SEE ALSO

krb5_auth_con_init(3), krb5_rd_priv(3), krb5_rd_safe(3)

NAME

krb5_initlog, **krb5_openlog**, **krb5_closelog**, **krb5_addlog_dest**, **krb5_addlog_func**, **krb5_log**, **krb5_vlog**, **krb5_log_msg**, **krb5_vlog_msg** — Heimdal logging functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

typedef void
(*krb5_log_log_func_t)(const char *time, const char *message, void *data);

typedef void
(*krb5_log_close_func_t)(void *data);

krb5_error_code
krb5_addlog_dest(krb5_context context, krb5_log_facility *facility,
    const char *destination);

krb5_error_code
krb5_addlog_func(krb5_context context, krb5_log_facility *facility,
    int min, int max, krb5_log_log_func_t log,
    krb5_log_close_func_t close, void *data);

krb5_error_code
krb5_closelog(krb5_context context, krb5_log_facility *facility);

krb5_error_code
krb5_initlog(krb5_context context, const char *program,
    krb5_log_facility **facility);

krb5_error_code
krb5_log(krb5_context context, krb5_log_facility *facility, int level,
    const char *format, ...);

krb5_error_code
krb5_log_msg(krb5_context context, krb5_log_facility *facility,
    char **reply, int level, const char *format, ...);

krb5_error_code
krb5_openlog(krb5_context context, const char *program,
    krb5_log_facility **facility);

krb5_error_code
krb5_vlog(krb5_context context, krb5_log_facility *facility, int level,
    const char *format, va_list arglist);

krb5_error_code
krb5_vlog_msg(krb5_context context, krb5_log_facility *facility,
    char **reply, int level, const char *format, va_list arglist);
```

DESCRIPTION

These functions logs messages to one or more destinations.

The **krb5_openlog**() function creates a logging *facility*, that is used to log messages. A facility consists of one or more destinations (which can be files or syslog or some other device). The *program* parameter should be the generic name of the program that is doing the logging. This name is used to lookup which

destinations to use. This information is contained in the `logging` section of the `krb5.conf` configuration file. If no entry is found for *`program`*, the entry for `default` is used, or if that is missing too, `SYSLOG` will be used as destination.

To close a logging facility, use the **krb5_closelog**() function.

To log a message to a facility use one of the functions **krb5_log**(), **krb5_log_msg**(), **krb5_vlog**(), or **krb5_vlog_msg**(). The functions ending in _msg return in *`reply`* a pointer to the message that just got logged. This string is allocated, and should be freed with **free**(). The *`format`* is a standard **printf**() style format string (but see the BUGS section).

If you want better control of where things gets logged, you can instead of using **krb5_openlog**() call **krb5_initlog**(), which just initializes a facility, but doesn't define any actual logging destinations. You can then add destinations with the **krb5_addlog_dest**() and **krb5_addlog_func**() functions. The first of these takes a string specifying a logging destination, and adds this to the facility. If you want to do some non-standard logging you can use the **krb5_addlog_func**() function, which takes a function to use when logging. The *`log`* function is called for each message with *`time`* being a string specifying the current time, and *`message`* the message to log. *`close`* is called when the facility is closed. You can pass application specific data in the *`data`* parameter. The *`min`* and *`max`* parameter are the same as in a destination (defined below). To specify a max of infinity, pass -1.

krb5_openlog() calls **krb5_initlog**() and then calls **krb5_addlog_dest**() for each destination found.

Destinations

The defined destinations (as specified in `krb5.conf`) follows:

`STDERR`
: This logs to the program's stderr.

`FILE:/file`

`FILE=/file`
: Log to the specified file. The form using a colon appends to the file, the form with an equal truncates the file. The truncating form keeps the file open, while the appending form closes it after each log message (which makes it possible to rotate logs). The truncating form is mainly for compatibility with the MIT libkrb5.

`DEVICE=/device`
: This logs to the specified device, at present this is the same as `FILE:/device`.

`CONSOLE`
: Log to the console, this is the same as `DEVICE=/dev/console`.

`SYSLOG`[:priority[:facility]]
: Send messages to the syslog system, using priority, and facility. To get the name for one of these, you take the name of the macro passed to `syslog`(3), and remove the leading `LOG_` (`LOG_NOTICE` becomes `NOTICE`). The default values (as well as the values used for unrecognised values), are `ERR`, and `AUTH`, respectively. See `syslog`(3) for a list of priorities and facilities.

Each destination may optionally be prepended with a range of logging levels, specified as `min-max/`. If the *`level`* parameter to **krb5_log**() is within this range (inclusive) the message gets logged to this destination, otherwise not. Either of the min and max valued may be omitted, in this case min is assumed to be zero, and max is assumed to be infinity. If you don't include a dash, both min and max gets set to the specified value. If no range is specified, all messages gets logged.

EXAMPLES

```
[logging]
        kdc = 0/FILE:/var/log/kdc.log
        kdc = 1-/SYSLOG:INFO:USER
        default = STDERR
```

This will log all messages from the **kdc** program with level 0 to `/var/log/kdc.log`, other messages will be logged to syslog with priority `LOG_INFO`, and facility `LOG_USER`. All other programs will log all messages to their stderr.

SEE ALSO

`syslog`(3), `krb5.conf`(5)

BUGS

These functions use **asprintf**() to format the message. If your operating system does not have a working **asprintf**(), a replacement will be used. At present this replacement does not handle some correct conversion specifications (like floating point numbers). Until this is fixed, the use of these conversions should be avoided.

If logging is done to the syslog facility, these functions might not be thread-safe, depending on the implementation of **openlog**(), and **syslog**().

NAME

krb5_parse_name — string to principal conversion

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_parse_name(krb5_context context, const char *name,
    krb5_principal *principal);
```

DESCRIPTION

krb5_parse_name() converts a string representation of a principal name to **krb5_principal**. The *principal* will point to allocated data that should be freed with **krb5_free_principal**().

The string should consist of one or more name components separated with slashes ("/"), optionally followed with an "@" and a realm name. A slash or @ may be contained in a name component by quoting it with a backslash ("\"). A realm should not contain slashes or colons.

SEE ALSO

krb5_425_conv_principal(3), krb5_build_principal(3), krb5_free_principal(3), krb5_sname_to_principal(3), krb5_unparse_name(3)

NAME

krb5_get_default_principal, **krb5_principal**, **krb5_build_principal**, **krb5_build_principal_ext**, **krb5_build_principal_va**, **krb5_build_principal_va_ext**, **krb5_copy_principal**, **krb5_free_principal**, **krb5_make_principal**, **krb5_parse_name**, **krb5_parse_name_flags**, **krb5_parse_nametype**, **krb5_princ_realm**, **krb5_princ_set_realm**, **krb5_principal_compare**, **krb5_principal_compare_any_realm**, **krb5_principal_get_comp_string**, **krb5_principal_get_realm**, **krb5_principal_get_type**, **krb5_principal_match**, **krb5_principal_set_type**, **krb5_realm_compare**, **krb5_sname_to_principal**, **krb5_sock_to_principal**, **krb5_unparse_name**, **krb5_unparse_name_flags**, **krb5_unparse_name_fixed**, **krb5_unparse_name_fixed_flags**, **krb5_unparse_name_fixed_short**, **krb5_unparse_name_short** — Kerberos 5 principal handling functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_principal;

void
krb5_free_principal(krb5_context context, krb5_principal principal);

krb5_error_code
krb5_parse_name(krb5_context context, const char *name,
    krb5_principal *principal);

krb5_error_code
krb5_parse_name_flags(krb5_context context, const char *name, int flags,
    krb5_principal *principal);

krb5_error_code
krb5_unparse_name(krb5_context context, krb5_const_principal principal,
    char **name);

krb5_error_code
krb5_unparse_name_flags(krb5_context context,
    krb5_const_principal principal, int flags, char **name);

krb5_error_code
krb5_unparse_name_fixed(krb5_context context,
    krb5_const_principal principal, char *name, size_t len);

krb5_error_code
krb5_unparse_name_fixed_flags(krb5_context context,
    krb5_const_principal principal, int flags, char *name, size_t len);

krb5_error_code
krb5_unparse_name_short(krb5_context context,
    krb5_const_principal principal, char **name);

krb5_error_code
krb5_unparse_name_fixed_short(krb5_context context,
    krb5_const_principal principal, char *name, size_t len);
```

krb5_realm ∗
krb5_princ_realm(*krb5_context context*, *krb5_principal principal*);

void
krb5_princ_set_realm(*krb5_context context*, *krb5_principal principal*,
krb5_realm ∗realm);

krb5_error_code
krb5_build_principal(*krb5_context context*, *krb5_principal ∗principal*,
int rlen, *krb5_const_realm realm*, *...*);

krb5_error_code
krb5_build_principal_va(*krb5_context context*, *krb5_principal ∗principal*,
int rlen, *krb5_const_realm realm*, *va_list ap*);

krb5_error_code
krb5_build_principal_ext(*krb5_context context*, *krb5_principal ∗principal*,
int rlen, *krb5_const_realm realm*, *...*);

krb5_error_code
krb5_build_principal_va_ext(*krb5_context context*,
krb5_principal ∗principal, *int rlen*, *krb5_const_realm realm*,
va_list ap);

krb5_error_code
krb5_make_principal(*krb5_context context*, *krb5_principal ∗principal*,
krb5_const_realm realm, *...*);

krb5_error_code
krb5_copy_principal(*krb5_context context*, *krb5_const_principal inprinc*,
krb5_principal ∗outprinc);

krb5_boolean
krb5_principal_compare(*krb5_context context*, *krb5_const_principal princ1*,
krb5_const_principal princ2);

krb5_boolean
krb5_principal_compare_any_realm(*krb5_context context*,
krb5_const_principal princ1, *krb5_const_principal princ2*);

const char ∗
krb5_principal_get_comp_string(*krb5_context context*,
krb5_const_principal principal, *unsigned int component*);

const char ∗
krb5_principal_get_realm(*krb5_context context*,
krb5_const_principal principal);

int
krb5_principal_get_type(*krb5_context context*,
krb5_const_principal principal);

krb5_boolean
krb5_principal_match(*krb5_context context*, *krb5_const_principal principal*,
krb5_const_principal pattern);

void
krb5_principal_set_type(*krb5_context context*, *krb5_principal principal*,
int type);

krb5_boolean
krb5_realm_compare(*krb5_context context*, *krb5_const_principal princ1*, *krb5_const_principal princ2*);

krb5_error_code
krb5_sname_to_principal(*krb5_context context*, *const char *hostname*, *const char *sname*, *int32_t type*, *krb5_principal *ret_princ*);

krb5_error_code
krb5_sock_to_principal(*krb5_context context*, *int socket*, *const char *sname*, *int32_t type*, *krb5_principal *principal*);

krb5_error_code
krb5_get_default_principal(*krb5_context context*, *krb5_principal *princ*);

krb5_error_code
krb5_parse_nametype(*krb5_context context*, *const char *str*, *int32_t *type*);

DESCRIPTION

`krb5_principal` holds the name of a user or service in Kerberos.

A principal has two parts, a `PrincipalName` and a `realm`. The PrincipalName consists of one or more components. In printed form, the components are separated by /. The PrincipalName also has a name-type.

Examples of a principal are `nisse/root@EXAMPLE.COM` and `host/datan.kth.se@KTH.SE`. **krb5_parse_name**() and **krb5_parse_name_flags**() passes a principal name in *name* to the kerberos principal structure. **krb5_parse_name_flags**() takes an extra *flags* argument the following flags can be passed in

`KRB5_PRINCIPAL_PARSE_NO_REALM`
: requries the input string to be without a realm, and no realm is stored in the *principal* return argument.

`KRB5_PRINCIPAL_PARSE_MUST_REALM`
: requries the input string to with a realm.

krb5_unparse_name() and **krb5_unparse_name_flags**() prints the principal *princ* to the string *name*. *name* should be freed with `free`(3). To the *flags* argument the following flags can be passed in

`KRB5_PRINCIPAL_UNPARSE_SHORT`
: no realm if the realm is one of the local realms.

`KRB5_PRINCIPAL_UNPARSE_NO_REALM`
: never include any realm in the principal name.

`KRB5_PRINCIPAL_UNPARSE_DISPLAY`
: don't quote

On failure *name* is set to `NULL`. **krb5_unparse_name_fixed**() and **krb5_unparse_name_fixed_flags**() behaves just like **krb5_unparse**(), but instead unparses the principal into a fixed size buffer.

krb5_unparse_name_short() just returns the principal without the realm if the principal is in the default realm. If the principal isn't, the full name is returned. **krb5_unparse_name_fixed_short**() works just like **krb5_unparse_name_short**() but on a fixed size buffer.

krb5_build_principal() builds a principal from the realm *realm* that has the length *rlen*. The following arguments form the components of the principal. The list of components is terminated with `NULL`.

krb5_build_principal_va() works like **krb5_build_principal**() using vargs.

krb5_build_principal_ext() and **krb5_build_principal_va_ext**() take a list of length-value pairs, the list is terminated with a zero length.

krb5_make_principal() works the same way as **krb5_build_principal**(), except it figures out the length of the realm itself.

krb5_copy_principal() makes a copy of a principal. The copy needs to be freed with **krb5_free_principal**().

krb5_principal_compare() compares the two principals, including realm of the principals and returns TRUE if they are the same and FALSE if not.

krb5_principal_compare_any_realm() works the same way as **krb5_principal_compare**() but doesn't compare the realm component of the principal.

krb5_realm_compare() compares the realms of the two principals and returns TRUE is they are the same, and FALSE if not.

krb5_principal_match() matches a *principal* against a *pattern*. The pattern is a globbing expression, where each component (separated by /) is matched against the corresponding component of the principal.

The **krb5_principal_get_realm**() and **krb5_principal_get_comp_string**() functions return parts of the *principal*, either the realm or a specific component. Both functions return string pointers to data inside the principal, so they are valid only as long as the principal exists.

The *component* argument to **krb5_principal_get_comp_string**() is the index of the component to return, from zero to the total number of components minus one. If the index is out of range NULL is returned.

krb5_principal_get_realm() and **krb5_principal_get_comp_string**() are replacements for **krb5_princ_realm**(), **krb5_princ_component**() and related macros, described as internal in the MIT API specification. Unlike the macros, these functions return strings, not krb5_data. A reason to return krb5_data was that it was believed that principal components could contain binary data, but this belief was unfounded, and it has been decided that principal components are infact UTF8, so it's safe to use zero terminated strings.

It's generally not necessary to look at the components of a principal.

krb5_principal_get_type() and **krb5_principal_set_type**() get and sets the name type for a principal. Name type handling is tricky and not often needed, don't use this unless you know what you do.

krb5_princ_realm() returns the realm component of the principal. The caller must not free realm unless **krb5_princ_set_realm**() is called to set a new realm after freeing the realm. **krb5_princ_set_realm**() sets the realm component of a principal. The old realm is not freed.

krb5_sname_to_principal() and **krb5_sock_to_principal**() are for easy creation of "service" principals that can, for instance, be used to lookup a key in a keytab. For both functions the *sname* parameter will be used for the first component of the created principal. If *sname* is NULL, "host" will be used instead.

krb5_sname_to_principal() will use the passed *hostname* for the second component. If *type* is KRB5_NT_SRV_HST this name will be looked up with **gethostbyname**(). If *hostname* is NULL, the local hostname will be used.

krb5_sock_to_principal() will use the "sockname" of the passed *socket*, which should be a bound AF_INET or AF_INET6 socket. There must be a mapping between the address and "sockname". The function may try to resolve the name in DNS.

krb5_get_default_principal() tries to find out what's a reasonable default principal by looking at the environment it is running in.

krb5_parse_nametype() parses and returns the name type integer value in *type*. On failure the function returns an error code and set the error string.

SEE ALSO

krb5_425_conv_principal(3), krb5_config(3), krb5.conf(5)

BUGS

You can not have a NUL in a component in some of the variable argument functions above. Until someone can give a good example of where it would be a good idea to have NUL's in a component, this will not be fixed.

NAME

krb5_rcache, **krb5_rc_close**, **krb5_rc_default**, **krb5_rc_default_name**, **krb5_rc_default_type**, **krb5_rc_destroy**, **krb5_rc_expunge**, **krb5_rc_get_lifespan**, **krb5_rc_get_name**, **krb5_rc_get_type**, **krb5_rc_initialize**, **krb5_rc_recover**, **krb5_rc_resolve**, **krb5_rc_resolve_full**, **krb5_rc_resolve_type**, **krb5_rc_store**, **krb5_get_server_rcache** — Kerberos 5 replay cache

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

struct krb5_rcache;

krb5_error_code
krb5_rc_close(krb5_context context, krb5_rcache id);

krb5_error_code
krb5_rc_default(krb5_context context, krb5_rcache *id);

const char *
krb5_rc_default_name(krb5_context context);

const char *
krb5_rc_default_type(krb5_context context);

krb5_error_code
krb5_rc_destroy(krb5_context context, krb5_rcache id);

krb5_error_code
krb5_rc_expunge(krb5_context context, krb5_rcache id);

krb5_error_code
krb5_rc_get_lifespan(krb5_context context, krb5_rcache id,
    krb5_deltat *auth_lifespan);

const char*
krb5_rc_get_name(krb5_context context, krb5_rcache id);

const char*
krb5_rc_get_type(krb5_context context, krb5_rcache id);

krb5_error_code
krb5_rc_initialize(krb5_context context, krb5_rcache id,
    krb5_deltat auth_lifespan);

krb5_error_code
krb5_rc_recover(krb5_context context, krb5_rcache id);

krb5_error_code
krb5_rc_resolve(krb5_context context, krb5_rcache id, const char *name);

krb5_error_code
krb5_rc_resolve_full(krb5_context context, krb5_rcache *id,
    const char *string_name);

krb5_error_code
krb5_rc_resolve_type(krb5_context context, krb5_rcache *id,
```

*const char *type*);

krb5_error_code
krb5_rc_store(*krb5_context context*, *krb5_rcache id*, *krb5_donot_replay *rep*);

krb5_error_code
krb5_get_server_rcache(*krb5_context context*, *const krb5_data *piece*,
*krb5_rcache *id*);

DESCRIPTION

The krb5_rcache structure holds a storage element that is used for data manipulation. The structure contains no public accessible elements.

krb5_rc_initialize() Creates the reply cache *id* and sets it lifespan to *auth_lifespan*. If the cache already exists, the content is destroyed.

SEE ALSO

krb5(3), krb5_data(3), kerberos(8)

NAME

krb5_rd_error, **krb5_free_error**, **krb5_free_error_contents**, **krb5_error_from_rd_error** — parse, free and read error from KRB-ERROR message

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_rd_error(krb5_context context, const krb5_data *msg,
    KRB_ERROR *result);

void
krb5_free_error(krb5_context context, krb5_error *error);

void
krb5_free_error_contents(krb5_context context, krb5_error *error);

krb5_error_code
krb5_error_from_rd_error(krb5_context context, const krb5_error *error,
    const krb5_creds *creds);
```

DESCRIPTION

Usually applications never needs to parse and understand Kerberos error messages since higher level functions will parse and push up the error in the krb5_context. These functions are described for completeness.

krb5_rd_error() parses and returns the kerboeros error message, the structure should be freed with **krb5_free_error_contents**() when the caller is done with the structure.

krb5_free_error() frees the content and the memory region holding the structure iself.

krb5_free_error_contents() free the content of the KRB-ERROR message.

krb5_error_from_rd_error() will parse the error message and set the error buffer in krb5_context to the error string passed back or the matching error code in the KRB-ERROR message. Caller should pick up the message with **krb5_get_error_string**(*3*) (don't forget to free the returned string with **krb5_free_error_string**()).

SEE ALSO

krb5(3), krb5_set_error_string(3), krb5_get_error_string(3), krb5.conf(5)

NAME

krb5_rd_safe, **krb5_rd_priv** — verifies authenticity of messages

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_rd_priv(krb5_context context, krb5_auth_context auth_context,
    const krb5_data *inbuf, krb5_data *outbuf, krb5_replay_data *outdata);

krb5_error_code
krb5_rd_safe(krb5_context context, krb5_auth_context auth_context,
    const krb5_data *inbuf, krb5_data *outbuf, krb5_replay_data *outdata);
```

DESCRIPTION

krb5_rd_safe() and **krb5_rd_priv**() parses KRB-SAFE and KRB-PRIV messages (as generated by krb5_mk_safe(3) and krb5_mk_priv(3)) from *inbuf* and verifies its integrity. The user data part of the message in put in *outbuf*. The encryption state, including keyblocks and addresses, is taken from *auth_context*. If the KRB5_AUTH_CONTEXT_RET_SEQUENCE or KRB5_AUTH_CONTEXT_RET_TIME flags are set in the *auth_context* the sequence number and time are returned in the *outdata* parameter.

SEE ALSO

krb5_auth_con_init(3), krb5_mk_priv(3), krb5_mk_safe(3)

NAME

krb5_copy_host_realm, **krb5_free_host_realm**, **krb5_get_default_realm**, **krb5_get_default_realms**, **krb5_get_host_realm**, **krb5_set_default_realm** — default and host realm read and manipulation routines

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_copy_host_realm(krb5_context context, const krb5_realm *from,
    krb5_realm **to);

krb5_error_code
krb5_free_host_realm(krb5_context context, krb5_realm *realmlist);

krb5_error_code
krb5_get_default_realm(krb5_context context, krb5_realm *realm);

krb5_error_code
krb5_get_default_realms(krb5_context context, krb5_realm **realm);

krb5_error_code
krb5_get_host_realm(krb5_context context, const char *host,
    krb5_realm **realms);

krb5_error_code
krb5_set_default_realm(krb5_context context, const char *realm);
```

DESCRIPTION

krb5_copy_host_realm() copies the list of realms from *from* to *to*. *to* should be freed by the caller using *krb5_free_host_realm*.

krb5_free_host_realm() frees all memory allocated by *realmlist*.

krb5_get_default_realm() returns the first default realm for this host. The realm returned should be freed with **free**().

krb5_get_default_realms() returns a NULL terminated list of default realms for this context. Realms returned by **krb5_get_default_realms**() should be freed with **krb5_free_host_realm**().

krb5_get_host_realm() returns a NULL terminated list of realms for *host* by looking up the information in the [domain_realm] in krb5.conf or in DNS. If the mapping in [domain_realm] results in the string dns_locate, DNS is used to lookup the realm.

When using DNS to a resolve the domain for the host a.b.c, **krb5_get_host_realm**() looks for a TXT resource record named _kerberos.a.b.c, and if not found, it strips off the first component and tries a again (_kerberos.b.c) until it reaches the root.

If there is no configuration or DNS information found, **krb5_get_host_realm**() assumes it can use the domain part of the *host* to form a realm. Caller must free *realmlist* with **krb5_free_host_realm**().

krb5_set_default_realm() sets the default realm for the *context*. If NULL is used as a *realm*, the [libdefaults]default_realm stanza in krb5.conf is used. If there is no such stanza in the con-

figuration file, the **krb5_get_host_realm**() function is used to form a default realm.

SEE ALSO

free(3), krb5.conf(5)

NAME

krb5_change_password, **krb5_set_password**, **krb5_set_password_using_ccache**, **krb5_passwd_result_to_string** — change password functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_change_password(krb5_context context, krb5_creds *creds, char *newpw,
    int *result_code, krb5_data *result_code_string,
    krb5_data *result_string);

krb5_error_code
krb5_set_password(krb5_context context, krb5_creds *creds, char *newpw,
    krb5_principal targprinc, int *result_code,
    krb5_data *result_code_string, krb5_data *result_string);

krb5_error_code
krb5_set_password_using_ccache(krb5_context context, krb5_ccache ccache,
    char *newpw, krb5_principal targprinc, int *result_code,
    krb5_data *result_code_string, krb5_data *result_string);

const char *
krb5_passwd_result_to_string(krb5_context context, int result);
```

DESCRIPTION

These functions change the password for a given principal.

krb5_set_password() and **krb5_set_password_using_ccache**() are the newer of the three functions, and use a newer version of the protocol (and also fall back to the older set-password protocol if the newer protocol doesn't work).

krb5_change_password() sets the password *newpasswd* for the client principal in *creds*. The server principal of creds must be kadmin/changepw.

krb5_set_password() and **krb5_set_password_using_ccache**() change the password for the principal *targprinc*.

krb5_set_password() requires that the credential for kadmin/changepw@REALM is in *creds*. If the user caller isn't an administrator, this credential needs to be an initial credential, see krb5_get_init_creds(3) how to get such credentials.

krb5_set_password_using_ccache() will get the credential from *ccache*.

If *targprinc* is NULL, **krb5_set_password_using_ccache**() uses the the default principal in *ccache* and **krb5_set_password**() uses the global the default principal.

All three functions return an error in *result_code* and maybe an error string to print in *result_string*.

krb5_passwd_result_to_string() returns an human readable string describing the error code in *result_code* from the **krb5_set_password**() functions.

SEE ALSO

`krb5_ccache`(3), `krb5_init_context`(3)

NAME

krb5_storage, **krb5_storage_emem**, **krb5_storage_from_data**, **krb5_storage_from_fd**, **krb5_storage_from_mem**, **krb5_storage_set_flags**, **krb5_storage_clear_flags**, **krb5_storage_is_flags**, **krb5_storage_set_byteorder**, **krb5_storage_get_byteorder**, **krb5_storage_set_eof_code**, **krb5_storage_seek**, **krb5_storage_read**, **krb5_storage_write**, **krb5_storage_free**, **krb5_storage_to_data**, **krb5_store_int32**, **krb5_ret_int32**, **krb5_store_uint32**, **krb5_ret_uint32**, **krb5_store_int16**, **krb5_ret_int16**, **krb5_store_uint16**, **krb5_ret_uint16**, **krb5_store_int8**, **krb5_ret_int8**, **krb5_store_uint8**, **krb5_ret_uint8**, **krb5_store_data**, **krb5_ret_data**, **krb5_store_string**, **krb5_ret_string**, **krb5_store_stringnl**, **krb5_ret_stringnl**, **krb5_store_stringz**, **krb5_ret_stringz**, **krb5_store_principal**, **krb5_ret_principal**, **krb5_store_keyblock**, **krb5_ret_keyblock**, **krb5_store_times**, **krb5_ret_times**, **krb5_store_address**, **krb5_ret_address**, **krb5_store_addrs**, **krb5_ret_addrs**, **krb5_store_authdata**, **krb5_ret_authdata**, **krb5_store_creds**, **krb5_ret_creds** — operates on the Kerberos datatype krb5_storage

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

struct krb5_storage;

krb5_storage *
krb5_storage_from_fd(int fd);

krb5_storage *
krb5_storage_emem(void);

krb5_storage *
krb5_storage_from_mem(void *buf, size_t len);

krb5_storage *
krb5_storage_from_data(krb5_data *data);

void
krb5_storage_set_flags(krb5_storage *sp, krb5_flags flags);

void
krb5_storage_clear_flags(krb5_storage *sp, krb5_flags flags);

krb5_boolean
krb5_storage_is_flags(krb5_storage *sp, krb5_flags flags);

void
krb5_storage_set_byteorder(krb5_storage *sp, krb5_flags byteorder);

krb5_flags
krb5_storage_get_byteorder(krb5_storage *sp, krb5_flags byteorder);

void
krb5_storage_set_eof_code(krb5_storage *sp, int code);
```

off_t
krb5_storage_seek(*krb5_storage *sp*, *off_t offset*, *int whence*);

krb5_ssize_t
krb5_storage_read(*krb5_storage *sp*, *void *buf*, *size_t len*);

krb5_ssize_t
krb5_storage_write(*krb5_storage *sp*, *const void *buf*, *size_t len*);

krb5_error_code
krb5_storage_free(*krb5_storage *sp*);

krb5_error_code
krb5_storage_to_data(*krb5_storage *sp*, *krb5_data *data*);

krb5_error_code
krb5_store_int32(*krb5_storage *sp*, *int32_t value*);

krb5_error_code
krb5_ret_int32(*krb5_storage *sp*, *int32_t *value*);

krb5_error_code
krb5_ret_uint32(*krb5_storage *sp*, *uint32_t *value*);

krb5_error_code
krb5_store_uint32(*krb5_storage *sp*, *uint32_t value*);

krb5_error_code
krb5_store_int16(*krb5_storage *sp*, *int16_t value*);

krb5_error_code
krb5_ret_int16(*krb5_storage *sp*, *int16_t *value*);

krb5_error_code
krb5_store_uint16(*krb5_storage *sp*, *uint16_t value*);

krb5_error_code
krb5_ret_uint16(*krb5_storage *sp*, *u_int16_t *value*);

krb5_error_code
krb5_store_int8(*krb5_storage *sp*, *int8_t value*);

krb5_error_code
krb5_ret_int8(*krb5_storage *sp*, *int8_t *value*);

krb5_error_code
krb5_store_uint8(*krb5_storage *sp*, *u_int8_t value*);

krb5_error_code
krb5_ret_uint8(*krb5_storage *sp*, *u_int8_t *value*);

krb5_error_code
krb5_store_data(*krb5_storage *sp*, *krb5_data data*);

krb5_error_code
krb5_ret_data(*krb5_storage *sp*, *krb5_data *data*);

krb5_error_code
krb5_store_string(*krb5_storage *sp*, *const char *s*);

krb5_error_code
krb5_ret_string(*krb5_storage *sp*, *char **string*);

krb5_error_code
krb5_store_stringnl(*krb5_storage *sp*, *const char *s*);

krb5_error_code
krb5_ret_stringnl(*krb5_storage *sp*, *char **string*);

krb5_error_code
krb5_store_stringz(*krb5_storage *sp*, *const char *s*);

krb5_error_code
krb5_ret_stringz(*krb5_storage *sp*, *char **string*);

krb5_error_code
krb5_store_principal(*krb5_storage *sp*, *krb5_const_principal p*);

krb5_error_code
krb5_ret_principal(*krb5_storage *sp*, *krb5_principal *princ*);

krb5_error_code
krb5_store_keyblock(*krb5_storage *sp*, *krb5_keyblock p*);

krb5_error_code
krb5_ret_keyblock(*krb5_storage *sp*, *krb5_keyblock *p*);

krb5_error_code
krb5_store_times(*krb5_storage *sp*, *krb5_times times*);

krb5_error_code
krb5_ret_times(*krb5_storage *sp*, *krb5_times *times*);

krb5_error_code
krb5_store_address(*krb5_storage *sp*, *krb5_address p*);

krb5_error_code
krb5_ret_address(*krb5_storage *sp*, *krb5_address *adr*);

krb5_error_code
krb5_store_addrs(*krb5_storage *sp*, *krb5_addresses p*);

krb5_error_code
krb5_ret_addrs(*krb5_storage *sp*, *krb5_addresses *adr*);

krb5_error_code
krb5_store_authdata(*krb5_storage *sp*, *krb5_authdata auth*);

krb5_error_code
krb5_ret_authdata(*krb5_storage *sp*, *krb5_authdata *auth*);

krb5_error_code
krb5_store_creds(*krb5_storage *sp*, *krb5_creds *creds*);

krb5_error_code
krb5_ret_creds(*krb5_storage *sp*, *krb5_creds *creds*);

DESCRIPTION

The `krb5_storage` structure holds a storage element that is used for data manipulation. The structure contains no public accessible elements.

krb5_storage_emem() create a memory based krb5 storage unit that dynamicly resized to the ammount of data stored in. The storage never returns errors, on memory allocation errors `exit`(3) will be called.

krb5_storage_from_data() create a krb5 storage unit that will read is data from a `krb5_data`. There is no copy made of the *data*, so the caller must not free *data* until the storage is freed.

krb5_storage_from_fd() create a krb5 storage unit that will read is data from a file descriptor. The descriptor must be seekable if **krb5_storage_seek**() is used. Caller must not free the file descriptor before the storage is freed.

krb5_storage_from_mem() create a krb5 storage unit that will read is data from a memory region. There is no copy made of the *data*, so the caller must not free *data* until the storage is freed.

krb5_storage_set_flags() and **krb5_storage_clear_flags**() modifies the behavior of the storage functions. **krb5_storage_is_flags**() tests if the *flags* are set on the `krb5_storage`. Valid flags to set, is and clear is are:

KRB5_STORAGE_PRINCIPAL_WRONG_NUM_COMPONENTS
: Stores the number of principal componets one too many when storing principal namees, used for compatibility with version 1 of file keytabs and version 1 of file credential caches.

KRB5_STORAGE_PRINCIPAL_NO_NAME_TYPE
: Doesn't store the name type in when storing a principal name, used for compatibility with version 1 of file keytabs and version 1 of file credential caches.

KRB5_STORAGE_KEYBLOCK_KEYTYPE_TWICE
: Stores the keyblock type twice storing a keyblock, used for compatibility version 3 of file credential caches.

KRB5_STORAGE_BYTEORDER_MASK
: bitmask that can be used to and out what type of byte order order is used.

KRB5_STORAGE_BYTEORDER_BE
: Store integers in in big endian byte order, this is the default mode.

KRB5_STORAGE_BYTEORDER_LE
: Store integers in in little endian byte order.

KRB5_STORAGE_BYTEORDER_HOST
: Stores the integers in host byte order, used for compatibility with version 1 of file keytabs and version 1 and 2 of file credential caches.

KRB5_STORAGE_CREDS_FLAGS_WRONG_BITORDER
: Store the credential flags in a krb5_creds in the reverse bit order.

krb5_storage_set_byteorder() and **krb5_storage_get_byteorder**() modifies the byte order used in the storage for integers. The flags used is same as above. The valid flags are `KRB5_STORAGE_BYTEORDER_BE`, `KRB5_STORAGE_BYTEORDER_LE` and `KRB5_STORAGE_BYTEORDER_HOST`.

krb5_storage_set_eof_code() sets the error code that will be returned on end of file condition to *code*.

krb5_storage_seek() seeks *offset* bytes in the storage *sp*. The *whence* argument is one of

SEEK_SET offset is from begining of storage.

SEEK_CUR
: offset is relative from current offset.

SEEK_END
: offset is from end of storage.

krb5_storage_read() reads *len* (or less bytes in case of end of file) into *buf* from the current offset in the storage *sp*.

krb5_storage_write() writes *len* or (less bytes in case of end of file) from *buf* from the current offset in the storage *sp*.

krb5_storage_free() frees the storage *sp*.

krb5_storage_to_data() converts the data in storage *sp* into a krb5_data structure. *data* must be freed with **krb5_data_free**() by the caller when done with the *data*.

All krb5_store and krb5_ret functions move the current offset forward when the functions returns.

krb5_store_int32(), **krb5_ret_int32**(), **krb5_store_uint32**(), **krb5_ret_uint32**(), **krb5_store_int16**(), **krb5_ret_int16**(), **krb5_store_uint16**(), **krb5_ret_uint16**(), **krb5_store_int8**(), **krb5_ret_int8**() **krb5_store_uint8**(), and **krb5_ret_uint8**() stores and reads an integer from *sp* in the byte order specified by the flags set on the *sp*.

krb5_store_data() and **krb5_ret_data**() store and reads a krb5_data. The length of the data is stored with **krb5_store_int32**().

krb5_store_string() and **krb5_ret_string**() store and reads a string by storing the length of the string with **krb5_store_int32**() followed by the string itself.

krb5_store_stringnl() and **krb5_ret_stringnl**() store and reads a string by storing string followed by a '0.

krb5_store_stringz() and **krb5_ret_stringz**() store and reads a string by storing string followed by a NUL.

krb5_store_principal() and **krb5_ret_principal**() store and reads a principal.

krb5_store_keyblock() and **krb5_ret_keyblock**() store and reads a krb5_keyblock.

krb5_store_times() **krb5_ret_times**() store and reads krb5_times structure .

krb5_store_address() and **krb5_ret_address**() store and reads a krb5_address.

krb5_store_addrs() and **krb5_ret_addrs**() store and reads a krb5_addresses.

krb5_store_authdata() and **krb5_ret_authdata**() store and reads a krb5_authdata.

krb5_store_creds() and **krb5_ret_creds**() store and reads a krb5_creds.

SEE ALSO

krb5(3), krb5_data(3), kerberos(8)

NAME

krb5_string_to_key, **krb5_string_to_key_data**, **krb5_string_to_key_data_salt**, **krb5_string_to_key_data_salt_opaque**, **krb5_string_to_key_salt**, **krb5_string_to_key_salt_opaque**, **krb5_get_pw_salt**, **krb5_free_salt** — turns a string to a Kerberos key

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_string_to_key(krb5_context context, krb5_enctype enctype,
    const char *password, krb5_principal principal, krb5_keyblock *key);

krb5_error_code
krb5_string_to_key_data(krb5_context context, krb5_enctype enctype,
    krb5_data password, krb5_principal principal, krb5_keyblock *key);

krb5_error_code
krb5_string_to_key_data_salt(krb5_context context, krb5_enctype enctype,
    krb5_data password, krb5_salt salt, krb5_keyblock *key);

krb5_error_code
krb5_string_to_key_data_salt_opaque(krb5_context context,
    krb5_enctype enctype, krb5_data password, krb5_salt salt,
    krb5_data opaque, krb5_keyblock *key);

krb5_error_code
krb5_string_to_key_salt(krb5_context context, krb5_enctype enctype,
    const char *password, krb5_salt salt, krb5_keyblock *key);

krb5_error_code
krb5_string_to_key_salt_opaque(krb5_context context, krb5_enctype enctype,
    const char *password, krb5_salt salt, krb5_data opaque,
    krb5_keyblock *key);

krb5_error_code
krb5_get_pw_salt(krb5_context context, krb5_const_principal principal,
    krb5_salt *salt);

krb5_error_code
krb5_free_salt(krb5_context context, krb5_salt salt);
```

DESCRIPTION

The string to key functions convert a string to a kerberos key.

krb5_string_to_key_data_salt_opaque() is the function that does all the work, the rest of the functions are just wrapers around **krb5_string_to_key_data_salt_opaque**() that calls it with default values.

krb5_string_to_key_data_salt_opaque() transforms the *password* with the given salt-string *salt* and the opaque, encryption type specific parameter *opaque* to a encryption key *key* according to the string to key function associated with *enctype*.

The *key* should be freed with **krb5_free_keyblock_contents**().

If one of the functions that doesn't take a krb5_salt as it argument **krb5_get_pw_salt**() is used to get the salt value.

krb5_get_pw_salt() get the default password salt for a principal, use **krb5_free_salt**() to free the salt when done.

krb5_free_salt() frees the content of *salt*.

SEE ALSO

krb5(3), krb5_data(3), krb5_keyblock(3), kerberos(8)

NAME

krb5_ticket, **krb5_free_ticket**, **krb5_copy_ticket**, **krb5_ticket_get_authorization_data_type**, **krb5_ticket_get_client**, **krb5_ticket_get_server**, **krb5_ticket_get_endtime** — Kerberos 5 ticket access and handling functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_ticket;

krb5_error_code
krb5_free_ticket(krb5_context context, krb5_ticket *ticket);

krb5_error_code
krb5_copy_ticket(krb5_context context, const krb5_ticket *from,
    krb5_ticket **to);

krb5_error_code
krb5_ticket_get_authorization_data_type(krb5_context context,
    krb5_ticket *ticket, int type, krb5_data *data);

krb5_error_code
krb5_ticket_get_client(krb5_context context, const krb5_ticket *ticket,
    krb5_principal *client);

krb5_error_code
krb5_ticket_get_server(krb5_context context, const krb5_ticket *ticket,
    krb5_principal *server);

time_t
krb5_ticket_get_endtime(krb5_context context, const krb5_ticket *ticket);
```

DESCRIPTION

`krb5_ticket` holds a kerberos ticket. The internals of the structure should never be accessed directly, functions exist for extracting information.

krb5_free_ticket() frees the *ticket* and its content. Used to free the result of **krb5_copy_ticket**() and **krb5_recvauth**().

krb5_copy_ticket() copies the content of the ticket *from* to the ticket *to*. The result *to* should be freed with **krb5_free_ticket**().

krb5_ticket_get_authorization_data_type() fetches the authorization data of the type *type* from the *ticket*. If there isn't any authorization data of type *type*, `ENOENT` is returned. *data* needs to be freed with **krb5_data_free**() on success.

krb5_ticket_get_client() and **krb5_ticket_get_server**() returns a copy of the client/server principal from the ticket. The principal returned should be free using `krb5_free_principal`(3).

krb5_ticket_get_endtime() return the end time of the ticket.

SEE ALSO

`krb5`(3)

NAME

krb5_timeofday, **krb5_set_real_time**, **krb5_us_timeofday**, **krb5_format_time**, **krb5_string_to_deltat** — Kerberos 5 time handling functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_timestamp;

krb5_deltat;

krb5_error_code
krb5_set_real_time(krb5_context context, krb5_timestamp sec, int32_t usec);

krb5_error_code
krb5_timeofday(krb5_context context, krb5_timestamp *timeret);

krb5_error_code
krb5_us_timeofday(krb5_context context, krb5_timestamp *sec, int32_t *usec);

krb5_error_code
krb5_format_time(krb5_context context, time_t t, char *s, size_t len,
    krb5_boolean include_time);

krb5_error_code
krb5_string_to_deltat(const char *string, krb5_deltat *deltat);
```

DESCRIPTION

krb5_set_real_time sets the absolute time that the caller knows the KDC has. With this the Kerberos library can calculate the relative difference between the KDC time and the local system time and store it in the *context*. With this information the Kerberos library can adjust all time stamps in Kerberos packages.

krb5_timeofday() returns the current time, but adjusted with the time difference between the local host and the KDC. **krb5_us_timeofday**() also returns microseconds.

krb5_format_time formats the time *t* into the string *s* of length *len*. If *include_time* is set, the time is set include_time.

krb5_string_to_deltat parses delta time *string* into *deltat*.

SEE ALSO

gettimeofday(2), krb5(3)

NAME

krb5_unparse_name — principal to string conversion

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_unparse_name(krb5_context context, krb5_principal principal,
    char **name);
```

DESCRIPTION

This function takes a *principal*, and will convert in to a printable representation with the same syntax as described in krb5_parse_name(3). **name* will point to allocated data and should be freed by the caller.

SEE ALSO

krb5_425_conv_principal(3), krb5_build_principal(3), krb5_free_principal(3), krb5_parse_name(3), krb5_sname_to_principal(3)

NAME

krb5_verify_init_creds_opt_init, **krb5_verify_init_creds_opt_set_ap_req_nofail**, **krb5_verify_init_creds** — verifies a credential cache is correct by using a local keytab

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

struct krb5_verify_init_creds_opt;

void
krb5_verify_init_creds_opt_init(krb5_verify_init_creds_opt *options);

void
krb5_verify_init_creds_opt_set_ap_req_nofail(krb5_verify_init_creds_opt *options,
    int ap_req_nofail);

krb5_error_code
krb5_verify_init_creds(krb5_context context, krb5_creds *creds,
    krb5_principal ap_req_server, krb5_ccache *ccache,
    krb5_verify_init_creds_opt *options);
```

DESCRIPTION

The **krb5_verify_init_creds** function verifies the initial tickets with the local keytab to make sure the response of the KDC was spoof-ed.

krb5_verify_init_creds will use principal *ap_req_server* from the local keytab, if NULL is passed in, the code will guess the local hostname and use that to form host/hostname/GUESSED-REALM-FOR-HOSTNAME. *creds* is the credential that **krb5_verify_init_creds** should verify. If *ccache* is given **krb5_verify_init_creds**() stores all credentials it fetched from the KDC there, otherwise it will use a memory credential cache that is destroyed when done.

krb5_verify_init_creds_opt_init() cleans the the structure, must be used before trying to pass it in to **krb5_verify_init_creds**().

krb5_verify_init_creds_opt_set_ap_req_nofail() controls controls the behavior if *ap_req_server* doesn't exists in the local keytab or in the KDC's database, if it's true, the error will be ignored. Note that this use is possible insecure.

SEE ALSO

krb5(3), krb5_get_init_creds(3), krb5_verify_user(3), krb5.conf(5)

NAME

krb5_verify_user, **krb5_verify_user_lrealm**, **krb5_verify_user_opt**, **krb5_verify_opt_init**, **krb5_verify_opt_alloc**, **krb5_verify_opt_free**, **krb5_verify_opt_set_ccache**, **krb5_verify_opt_set_flags**, **krb5_verify_opt_set_service**, **krb5_verify_opt_set_secure**, **krb5_verify_opt_set_keytab** — Heimdal password verifying functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_verify_user(krb5_context context, krb5_principal principal,
    krb5_ccache ccache, const char *password, krb5_boolean secure,
    const char *service);

krb5_error_code
krb5_verify_user_lrealm(krb5_context context, krb5_principal principal,
    krb5_ccache ccache, const char *password, krb5_boolean secure,
    const char *service);

void
krb5_verify_opt_init(krb5_verify_opt *opt);

void
krb5_verify_opt_alloc(krb5_verify_opt **opt);

void
krb5_verify_opt_free(krb5_verify_opt *opt);

void
krb5_verify_opt_set_ccache(krb5_verify_opt *opt, krb5_ccache ccache);

void
krb5_verify_opt_set_keytab(krb5_verify_opt *opt, krb5_keytab keytab);

void
krb5_verify_opt_set_secure(krb5_verify_opt *opt, krb5_boolean secure);

void
krb5_verify_opt_set_service(krb5_verify_opt *opt, const char *service);

void
krb5_verify_opt_set_flags(krb5_verify_opt *opt, unsigned int flags);

krb5_error_code
krb5_verify_user_opt(krb5_context context,krb5_principal principal,
    const char *password,krb5_verify_opt *opt);
```

DESCRIPTION

The **krb5_verify_user** function verifies the password supplied by a user. The principal whose password will be verified is specified in *principal*. New tickets will be obtained as a side-effect and stored in *ccache* (if NULL, the default ccache is used). **krb5_verify_user**() will call **krb5_cc_initialize**() on the given *ccache*, so *ccache* must only initialized with **krb5_cc_resolve**() or **krb5_cc_gen_new**(). If the password is not supplied in *password* (and is

given as NULL) the user will be prompted for it. If *secure* the ticket will be verified against the locally stored service key *service* (by default host if given as NULL).

The **krb5_verify_user_lrealm**() function does the same, except that it ignores the realm in *principal* and tries all the local realms (see krb5.conf(5)). After a successful return, the principal is set to the authenticated realm. If the call fails, the principal will not be meaningful, and should only be freed with krb5_free_principal(3).

krb5_verify_opt_alloc() and **krb5_verify_opt_free**() allocates and frees a krb5_verify_opt. You should use the the alloc and free function instead of allocation the structure yourself, this is because in a future release the structure wont be exported.

krb5_verify_opt_init() resets all opt to default values.

None of the krb5_verify_opt_set function makes a copy of the data structure that they are called with. It's up the caller to free them after the **krb5_verify_user_opt**() is called.

krb5_verify_opt_set_ccache() sets the *ccache* that user of *opt* will use. If not set, the default credential cache will be used.

krb5_verify_opt_set_keytab() sets the *keytab* that user of *opt* will use. If not set, the default keytab will be used.

krb5_verify_opt_set_secure() if *secure* if true, the password verification will require that the ticket will be verified against the locally stored service key. If not set, default value is true.

krb5_verify_opt_set_service() sets the *service* principal that user of *opt* will use. If not set, the host service will be used.

krb5_verify_opt_set_flags() sets *flags* that user of *opt* will use. If the flag KRB5_VERIFY_LREALMS is used, the *principal* will be modified like **krb5_verify_user_lrealm**() modifies it.

krb5_verify_user_opt() function verifies the *password* supplied by a user. The principal whose password will be verified is specified in *principal*. Options the to the verification process is pass in in *opt*.

EXAMPLES

Here is a example program that verifies a password. it uses the host/'hostname' service principal in krb5.keytab.

```
#include <krb5/krb5.h>

int
main(int argc, char **argv)
{
    char *user;
    krb5_error_code error;
    krb5_principal princ;
    krb5_context context;

    if (argc != 2)
        errx(1, "usage: verify_passwd <principal-name>");

    user = argv[1];

    if (krb5_init_context(&context) < 0)
```

```
            errx(1, "krb5_init_context");

        if ((error = krb5_parse_name(context, user, &princ)) != 0)
            krb5_err(context, 1, error, "krb5_parse_name");

        error = krb5_verify_user(context, princ, NULL, NULL, TRUE, NULL);
        if (error)
            krb5_err(context, 1, error, "krb5_verify_user");

        return 0;
    }
```

SEE ALSO

krb5_cc_gen_new(3), krb5_cc_initialize(3), krb5_cc_resolve(3), krb5_err(3), krb5_free_principal(3), krb5_init_context(3), krb5_kt_default(3), krb5.conf(5)

NAME

krb5_abort, **krb5_abortx**, **krb5_clear_error_string**, **krb5_err**, **krb5_errx**, **krb5_free_error_string**, **krb5_get_err_text**, **krb5_get_error_message**, **krb5_get_error_string**, **krb5_have_error_string**, **krb5_set_error_string**, **krb5_set_warn_dest**, **krb5_get_warn_dest**, **krb5_vabort**, **krb5_vabortx**, **krb5_verr**, **krb5_verrx**, **krb5_vset_error_string**, **krb5_vwarn**, **krb5_vwarnx**, **krb5_warn**, **krb5_warnx** — Heimdal warning and error functions

LIBRARY

Kerberos 5 Library (libkrb5, -lkrb5)

SYNOPSIS

```
#include <krb5/krb5.h>

krb5_error_code
krb5_abort(krb5_context context, krb5_error_code code, const char *fmt,
    ...);

krb5_error_code
krb5_abortx(krb5_context context, krb5_error_code code, const char *fmt,
    ...);

void
krb5_clear_error_string(krb5_context context);

krb5_error_code
krb5_err(krb5_context context, int eval, krb5_error_code code,
    const char *format, ...);

krb5_error_code
krb5_errx(krb5_context context, int eval, const char *format, ...);

void
krb5_free_error_string(krb5_context context, char *str);

krb5_error_code
krb5_verr(krb5_context context, int eval, krb5_error_code code,
    const char *format, va_list ap);

krb5_error_code
krb5_verrx(krb5_context context, int eval, const char *format, va_list ap);

krb5_error_code
krb5_vset_error_string(krb5_context context, const char *fmt,
    va_list args);

krb5_error_code
krb5_vwarn(krb5_context context, krb5_error_code code, const char *format,
    va_list ap);

krb5_error_code
krb5_vwarnx(krb5_context context, const char *format, va_list ap);

krb5_error_code
krb5_warn(krb5_context context, krb5_error_code code, const char *format,
    ...);
```

krb5_error_code
krb5_warnx(*krb5_context context*, *const char *format*, *...*);

krb5_error_code
krb5_set_error_string(*krb5_context context*, *const char *fmt*, *...*);

krb5_error_code
krb5_set_warn_dest(*krb5_context context*, *krb5_log_facility *facility*);

*char **

*krb5_log_facility **
krb5_get_warn_dest(*krb5_context context*);

krb5_get_err_text(*krb5_context context*, *krb5_error_code code*);

*char**
krb5_get_error_string(*krb5_context context*);

*char**
krb5_get_error_message(*krb5_context context, krb5_error_code code*);

krb5_boolean
krb5_have_error_string(*krb5_context context*);

krb5_error_code
krb5_vabortx(*krb5_context context*, *const char *fmt*, *va_list ap*);

krb5_error_code
krb5_vabort(*krb5_context context*, *const char *fmt*, *va_list ap*);

DESCRIPTION

These functions print a warning message to some destination. *format* is a printf style format specifying the message to print. The forms not ending in an "x" print the error string associated with *code* along with the message. The "err" functions exit with exit status *eval* after printing the message.

Applications that want to get the error message to report it to a user or store it in a log want to use **krb5_get_error_message**().

The **krb5_set_warn_func**() function sets the destination for warning messages to the specified *facility*. Messages logged with the "warn" functions have a log level of 1, while the "err" functions log with level 0.

krb5_get_err_text() fetches the human readable strings describing the error-code.

krb5_abort() and **krb5_abortx** behaves like **krb5_err** and **krb5_errx** but instead of exiting using the exit(3) call, abort(3) is used.

krb5_free_error_string() frees the error string *str* returned by **krb5_get_error_string**().

krb5_clear_error_string() clears the error string from the *context*.

krb5_set_error_string() and **krb5_vset_error_string**() sets an verbose error string in *context*.

krb5_get_error_string() fetches the error string from *context*. The error message in the context is consumed and must be freed using **krb5_free_error_string**() by the caller. See also **krb5_get_error_message**(), what is usually less verbose to use.

krb5_have_error_string() returns TRUE if there is a verbose error message in the *context*.

krb5_get_error_message() fetches the error string from the context, or if there is no customized error string in *context*, uses *code* to return a error string. In either case, the error message in the context is consumed and must be freed using **krb5_free_error_string**() by the caller.

krb5_set_warn_dest() and **krb5_get_warn_dest**() sets and get the log context that is used by **krb5_warn**() and friends. By using this the application can control where the output should go. For example, this is imperative to inetd servers where logging status and error message will end up on the output stream to the client.

EXAMPLES

Below is a simple example how to report error messages from the Kerberos library in an application.

```
#include <krb5/krb5.h>

krb5_error_code
function (krb5_context context)
{
    krb5_error_code ret;

    ret = krb5_function (context, arg1, arg2);
    if (ret) {
        char *s = krb5_get_error_message(context, ret);
        if (s == NULL)
             errx(1, "kerberos error: %d (and out of memory)", ret);
        application_logger("krb5_function failed: %s", s);
        krb5_free_error_string(context, s);
        return ret;
    }
    return 0;
}
```

SEE ALSO

krb5(3), krb5_openlog(3)

NAME

kvm — kernel memory interface

LIBRARY

Kernel Data Access Library (libkvm, −lkvm)

DESCRIPTION

The **kvm** library provides a uniform interface for accessing kernel virtual memory images, including live systems and crash dumps. Access to live systems is via `/dev/mem` while crash dumps can be examined via the core file generated by `savecore`(8). The interface behaves identically in both cases. Memory can be read and written, kernel symbol addresses can be looked up efficiently, and information about user processes can be gathered.

kvm_open() is first called to obtain a descriptor for all subsequent calls.

COMPATIBILITY

The kvm interface was first introduced in SunOS. A considerable number of programs have been developed that use this interface, making backward compatibility highly desirable. In most respects, the Sun kvm interface is consistent and clean. Accordingly, the generic portion of the interface (i.e., **kvm_open**(), **kvm_close**(), **kvm_read**(), **kvm_write**(), and **kvm_nlist**()) has been incorporated into the BSD interface. Indeed, many kvm applications (i.e., debuggers and statistical monitors) use only this subset of the interface.

The process interface was not kept. This is not a portability issue since any code that manipulates processes is inherently machine dependent.

Finally, the Sun kvm error reporting semantics are poorly defined. The library can be configured either to print errors to stderr automatically, or to print no error messages at all. In the latter case, the nature of the error cannot be determined. To overcome this, the BSD interface includes a routine, `kvm_geterr`(3), to return (not print out) the error message corresponding to the most recent error condition on the given descriptor.

FILES

`/dev/mem` interface to physical memory

SEE ALSO

`kvm_close`(3), `kvm_getargv`(3), `kvm_getenvv`(3), `kvm_geterr`(3), `kvm_getloadavg`(3), `kvm_getlwps`(3), `kvm_getprocs`(3), `kvm_nlist`(3), `kvm_open`(3), `kvm_openfiles`(3), `kvm_read`(3), `kvm_write`(3)

NAME

kvm_dump_mkheader, **kvm_dump_wrtheader**, **kvm_dump_inval** — crash dump support functions

LIBRARY

Kernel Data Access Library (libkvm, −lkvm)

SYNOPSIS

```
#include <kvm.h>

int
kvm_dump_mkheader(kvm_t *kd, off_t dump_off);

int
kvm_dump_wrtheader(kvm_t *kd, FILE *fp, int dumpsize);

int
kvm_dump_inval(kvm_t *kd);
```

DESCRIPTION

First note that the functions described here were designed to be used by savecore(8).

The function **kvm_dump_mkheader**() checks if the physical memory file associated with *kd* contains a valid crash dump header as generated by a dumping kernel. When a valid header is found, **kvm_dump_mkheader**() initializes the internal kvm data structures as if a crash dump generated by the savecore(8) program was opened. This has the intentional side effect of enabling the address translation machinery.

A call to **kvm_dump_mkheader**() will most likely be followed by a call to **kvm_dump_wrtheader**(). This function takes care of generating the generic header, the CORE_CPU section and the section header of the CORE_DATA section. The data is written to the file pointed at by *fp*. The *dumpsize* argument is only used to properly the set the segment size of the CORE_DATA section. Note that this function assumes that *fp* is positioned at file location 0. This function will not seek and therefore allows *fp* to be a file pointer obtained by **zopen**().

The **kvm_dump_inval**() function clears the magic number in the physical memory file associated with *kd*. The address translations must be enabled for this to work (thus assuming that **kvm_dump_mkheader**() was called earlier in the sequence).

RETURN VALUES

All functions except **kvm_dump_mkheader**() return 0 on success, -1 on failure. The function **kvm_dump_mkheader**() returns the size of the headers present before the actual dumpdata starts. If no valid headers were found but no fatal errors occurred, 0 is returned. On fatal errors the return value is -1.

In the case of failure, kvm_geterr(3) can be used to retrieve the cause of the error.

SEE ALSO

kvm(3), kvm_open(3)

HISTORY

These functions first appeared in NetBSD 1.2.

NAME

kvm_geterr — get error message on kvm descriptor

LIBRARY

Kernel Data Access Library (libkvm, −lkvm)

SYNOPSIS

```
#include <kvm.h>

char *
kvm_geterr(kvm_t *kd);
```

DESCRIPTION

This function returns a string describing the most recent error condition on the descriptor *kd*. The results are undefined if the most recent kvm(3) library call did not produce an error. The string returned is stored in memory owned by kvm(3) so the message should be copied out and saved elsewhere if necessary.

SEE ALSO

kvm(3), kvm_close(3), kvm_getargv(3), kvm_getenvv(3), kvm_getprocs(3), kvm_nlist(3), kvm_open(3), kvm_openfiles(3), kvm_read(3), kvm_write(3)

BUGS

This routine cannot be used to access error conditions due to a failed **kvm_openfiles**() call, since failure is indicated by returning a NULL descriptor. Therefore, errors on open are output to the special error buffer passed to **kvm_openfiles**(). This option is not available to **kvm_open**().

NAME

kvm_getfiles — survey open files

LIBRARY

Kernel Data Access Library (libkvm, −lkvm)

SYNOPSIS

```
#include <kvm.h>
#include <sys/kinfo.h>
#define _KERNEL
#include <sys/file.h>
#undef _KERNEL

char *
kvm_getfiles(kvm_t *kd, int op, int arg, int *cnt);
```

DESCRIPTION

kvm_getfiles() returns a (sub-)set of the open files in the kernel indicated by *kd*. The *op* and *arg* arguments constitute a predicate which limits the set of files returned. No predicates are currently defined.

The number of processes found is returned in the reference parameter *cnt*. The files are returned as a contiguous array of file structures, preceded by the address of the first file entry in the kernel. This memory is owned by kvm and is not guaranteed to be persistent across subsequent kvm library calls. Data should be copied out if it needs to be saved.

RETURN VALUES

kvm_getfiles() will return NULL on failure.

SEE ALSO

kvm(3), kvm_close(3), kvm_geterr(3), kvm_nlist(3), kvm_open(3), kvm_openfiles(3), kvm_read(3), kvm_write(3)

BUGS

This routine does not belong in the kvm interface.

NAME

kvm_getloadavg — get system load averages

LIBRARY

Kernel Data Access Library (libkvm, −lkvm)

SYNOPSIS

```
#include <sys/resource.h>
#include <kvm.h>

int
kvm_getloadavg(kvm_t *kd, double loadavg[], int nelem);
```

DESCRIPTION

The **kvm_getloadavg**() function returns the number of processes in the system run queue of the kernel indicated by *kd*, averaged over various periods of time. Up to *nelem* samples are retrieved and assigned to successive elements of *loadavg*[]. The system imposes a maximum of 3 samples, representing averages over the last 1, 5, and 15 minutes, respectively.

RETURN VALUES

If the load average was unobtainable, −1 is returned; otherwise, the number of samples actually retrieved is returned.

SEE ALSO

uptime(1), getloadavg(3), kvm(3), kvm_open(3), sysctl(3)

NAME

kvm_getlwps — access state of LWPs belonging to a user process

LIBRARY

Kernel Data Access Library (libkvm, −lkvm)

SYNOPSIS

```
#include <kvm.h>
#include <sys/param.h>
#include <sys/sysctl.h>

struct kinfo_lwp *
kvm_getlwps(kvm_t *kd, int pid, u_long procaddr, int *elemsize, int *cnt);
```

DESCRIPTION

kvm_getlwps() returns the set of LWPs belonging to the process specified by *pid* or *procaddr* in the kernel indicated by *kd*. The number of LWPs found is returned in the reference parameter *cnt*. The LWPs are returned as a contiguous array of **kinfo_lwp** structures. This memory is locally allocated, and subsequent calls to **kvm_getlwps**() and **kvm_close**() will overwrite this storage.

Only the first *elemsize* bytes of each array entry are returned. If the size of the **kinfo_lwp** structure increases in size in a future release of NetBSD the kernel will only return the requested amount of data for each array entry and programs that use **kvm_getlwps**() will continue to function without the need for recompilation.

If called against an active kernel, the **kvm_getlwps**() function will use the sysctl(3) interface and return information about the process identified by *pid*; otherwise the kernel memory device file or swap device will be accessed and the process is identified by the location passed in *paddr*.

RETURN VALUES

kvm_getlwps() returns NULL on failure.

SEE ALSO

kvm(3), kvm_close(3), kvm_geterr(3), kvm_getproc2(3), kvm_getprocs(3), kvm_nlist(3), kvm_open(3), kvm_openfiles(3), kvm_read(3), kvm_write(3)

BUGS

These routines do not belong in the kvm interface.

NAME

kvm_getprocs, **kvm_getargv**, **kvm_getenvv** — access user process state

LIBRARY

Kernel Data Access Library (libkvm, −lkvm)

SYNOPSIS

```
#include <kvm.h>
#include <sys/param.h>
#include <sys/sysctl.h>

struct kinfo_proc *
kvm_getprocs(kvm_t *kd, int op, int arg, int *cnt);

char **
kvm_getargv(kvm_t *kd, const struct kinfo_proc *p, int nchr);

char **
kvm_getenvv(kvm_t *kd, const struct kinfo_proc *p, int nchr);

struct kinfo_proc2 *
kvm_getproc2(kvm_t *kd, int op, int arg, int elemsize, int *cnt);

char **
kvm_getargv2(kvm_t *kd, const struct kinfo_proc2 *p, int nchr);

char **
kvm_getenvv2(kvm_t *kd, const struct kinfo_proc2 *p, int nchr);
```

DESCRIPTION

kvm_getprocs() returns a (sub-)set of active processes in the kernel indicated by *kd*. The *op* and *arg* arguments constitute a predicate which limits the set of processes returned. The value of *op* describes the filtering predicate as follows:

KERN_PROC_ALL	all processes
KERN_PROC_PID	processes with process id *arg*
KERN_PROC_PGRP	processes with process group *arg*
KERN_PROC_SESSION	processes with session id *arg*
KERN_PROC_TTY	processes with tty device *arg*
KERN_PROC_UID	processes with effective user id *arg*
KERN_PROC_RUID	processes with real user id *arg*
KERN_PROC_GID	processes with effective group id *arg*
KERN_PROC_RGID	processes with real group id *arg*

The number of processes found is returned in the reference parameter *cnt*. The processes are returned as a contiguous array of **kinfo_proc** structures. This memory is locally allocated, and subsequent calls to **kvm_getprocs**() and **kvm_close**() will overwrite this storage.

If the *op* argument for **kvm_getprocs**() is **KERN_PROC_TTY**, *arg* can also be **KERN_PROC_TTY_NODEV** to select processes with no controlling tty and **KERN_PROC_TTY_REVOKE** to select processes which have had their controlling tty revoked.

kvm_getargv() returns a null-terminated argument vector that corresponds to the command line arguments passed to process indicated by *p*. Most likely, these arguments correspond to the values passed to exec(3) on process creation. This information is, however, deliberately under control of the process itself. Note that the original command name can be found, unaltered, in the p_comm field of the process structure

returned by **kvm_getprocs**().

The *nchr* argument indicates the maximum number of characters, including null bytes, to use in building the strings. If this amount is exceeded, the string causing the overflow is truncated and the partial result is returned. This is handy for programs like ps(1) and w(1) that print only a one line summary of a command and should not copy out large amounts of text only to ignore it. If *nchr* is zero, no limit is imposed and all argument strings are returned in their entirety.

The memory allocated to the argv pointers and string storage is owned by the kvm library. Subsequent **kvm_getprocs**() and kvm_close(3) calls will clobber this storage.

The **kvm_getenvv**() function is similar to **kvm_getargv**() but returns the vector of environment strings. This data is also alterable by the process.

kvm_getproc2() is similar to **kvm_getprocs**() but returns an array of **kinfo_proc2** structures. Additionally, only the first *elemsize* bytes of each array entry are returned. If the size of the **kinfo_proc2** structure increases in size in a future release of NetBSD the kernel will only return the requested amount of data for each array entry and programs that use **kvm_getproc2**() will continue to function without the need for recompilation.

The **kvm_getargv2**() and **kvm_getenvv2**() are equivalents to the **kvm_getargv**() and **kvm_getenvv**() functions but use a **kinfo_proc2** structure to specify the process.

If called against an active kernel, the **kvm_getproc2**(), **kvm_getargv2**(), and **kvm_getenvv2**() functions will use the sysctl(3) interface and do not require access to the kernel memory device file or swap device.

RETURN VALUES

kvm_getprocs(), **kvm_getargv**(), **kvm_getenvv**(), **kvm_getproc2**(), **kvm_getargv2**(), and **kvm_getenvv2**() all return NULL on failure.

SEE ALSO

kvm(3), kvm_close(3), kvm_geterr(3), kvm_nlist(3), kvm_open(3), kvm_openfiles(3), kvm_read(3), kvm_write(3)

BUGS

These routines do not belong in the kvm interface.

NAME

kvm_nlist — retrieve symbol table names from a kernel image

LIBRARY

Kernel Data Access Library (libkvm, −lkvm)

SYNOPSIS

```
#include <kvm.h>
#include <nlist.h>

int
kvm_nlist(kvm_t *kd, struct nlist *nl);
```

DESCRIPTION

kvm_nlist() retrieves the symbol table entries indicated by the name list argument *nl*. This argument points to an array of nlist structures, terminated by an entry whose n_name field is NULL (see nlist(3)). Each symbol is looked up using the n_name field, and if found, the corresponding n_type and n_value fields are filled in. These fields are set to 0 if the symbol is not found.

If *kd* was created by a call to **kvm_open**() with a NULL executable image name, **kvm_nlist**() will use /dev/ksyms to retrieve the kernel symbol table.

RETURN VALUES

The **kvm_nlist**() function returns the number of invalid entries found. If the kernel symbol table was unreadable, -1 is returned.

FILES

/dev/ksyms

SEE ALSO

kvm(3), kvm_close(3), kvm_getargv(3), kvm_getenvv(3), kvm_geterr(3), kvm_getprocs(3), kvm_open(3), kvm_openfiles(3), kvm_read(3), kvm_write(3), ksyms(4)

NAME

kvm_open, **kvm_openfiles**, **kvm_close** — initialize kernel virtual memory access

LIBRARY

Kernel Data Access Library (libkvm, −lkvm)

SYNOPSIS

```
#include <fcntl.h>
#include <kvm.h>

kvm_t *
kvm_open(const char *execfile, const char *corefile, char *swapfile,
    int flags, const char *errstr);

kvm_t *
kvm_openfiles(const char *execfile, const char *corefile, char *swapfile,
    int flags, char *errbuf);

int
kvm_close(kvm_t *kd);
```

DESCRIPTION

The functions **kvm_open**() and **kvm_openfiles**() return a descriptor used to access kernel virtual memory via the kvm(3) library routines. Both active kernels and crash dumps are accessible through this interface.

execfile is the executable image of the kernel being examined. This file must contain a symbol table. If this argument is NULL, the currently running system is assumed, which is indicated by _PATH_UNIX in ⟨paths.h⟩.

corefile is the kernel memory device file. It can be either /dev/mem or a crash dump core generated by savecore(8). If *corefile* is NULL, the default indicated by _PATH_MEM from ⟨paths.h⟩ is used.

swapfile should indicate the swap device. If NULL, _PATH_DRUM from ⟨paths.h⟩ is used.

The *flags* argument indicates read/write access as in open(2) and applies only to the core file. The only permitted flags from open(2) are O_RDONLY, O_WRONLY, and O_RDWR.

As a special case, a *flags* argument of KVM_NO_FILES will initialize the kvm(3) library for use on active kernels only using sysctl(3) for retrieving kernel data and ignores the *execfile*, *corefile* and *swapfile* arguments. Only a small subset of the kvm(3) library functions are available using this method. These are currently kvm_getproc2(3), kvm_getargv2(3) and kvm_getenvv2(3).

There are two open routines which differ only with respect to the error mechanism. One provides backward compatibility with the SunOS kvm library, while the other provides an improved error reporting framework.

The **kvm_open**() function is the Sun kvm compatible open call. Here, the *errstr* argument indicates how errors should be handled. If it is NULL, no errors are reported and the application cannot know the specific nature of the failed kvm call. If it is not NULL, errors are printed to stderr with *errstr* prepended to the message, as in perror(3). Normally, the name of the program is used here. The string is assumed to persist at least until the corresponding **kvm_close**() call.

The **kvm_openfiles**() function provides BSD style error reporting. Here, error messages are not printed out by the library. Instead, the application obtains the error message corresponding to the most recent kvm library call using **kvm_geterr**() (see kvm_geterr(3)). The results are undefined if the most recent kvm call did not produce an error. Since **kvm_geterr**() requires a kvm descriptor, but the open routines return NULL on failure, **kvm_geterr**() cannot be used to get the error message if open fails. Thus,

kvm_openfiles() will place any error message in the *errbuf* argument. This buffer should be _POSIX2_LINE_MAX characters large (from ⟨limits.h⟩).

RETURN VALUES

The **kvm_open**() and **kvm_openfiles**() functions both return a descriptor to be used in all subsequent kvm library calls. The library is fully re-entrant. On failure, NULL is returned, in which case **kvm_openfiles**() writes the error message into *errbuf*.

The **kvm_close**() function returns 0 on success and -1 on failure.

SEE ALSO

open(2), kvm(3), kvm_getargv(3), kvm_getenvv(3), kvm_geterr(3), kvm_getprocs(3), kvm_nlist(3), kvm_read(3), kvm_write(3)

BUGS

There should not be two open calls. The ill-defined error semantics of the Sun library and the desire to have a backward-compatible library for BSD left little choice.

NAME

kvm_read, **kvm_write** — read or write kernel virtual memory

LIBRARY

Kernel Data Access Library (libkvm, −lkvm)

SYNOPSIS

```
#include <kvm.h>

ssize_t
kvm_read(kvm_t *kd, u_long addr, void *buf, size_t nbytes);

ssize_t
kvm_write(kvm_t *kd, u_long addr, const void *buf, size_t nbytes);
```

DESCRIPTION

The **kvm_read**() and **kvm_write**() functions are used to read and write kernel virtual memory (or a crash dump file). See **kvm_open**(*3*) or **kvm_openfiles**(*3*) for information regarding opening kernel virtual memory and crash dumps.

The **kvm_read**() function transfers *nbytes* bytes of data from the kernel space address *addr* to *buf*. Conversely, **kvm_write**() transfers data from *buf* to *addr*. Unlike their SunOS counterparts, these functions cannot be used to read or write process address spaces.

RETURN VALUES

Upon success, the number of bytes actually transferred is returned. Otherwise, -1 is returned.

SEE ALSO

kvm(3), kvm_close(3), kvm_getargv(3), kvm_getenvv(3), kvm_geterr(3), kvm_getprocs(3), kvm_nlist(3), kvm_open(3), kvm_openfiles(3)

NAME

labs — return the absolute value of a long integer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

long int
labs(long int j);
```

DESCRIPTION

The **labs**() function returns the absolute value of the long integer *j*.

SEE ALSO

abs(3), cabs(3), floor(3), llabs(3), math(3)

STANDARDS

The **labs**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

BUGS

The absolute value of the most negative integer remains negative.

NAME

ber_get_next, ber_skip_tag, ber_peek_tag, ber_scanf, ber_get_int, ber_get_enum, ber_get_stringb, ber_get_stringa, ber_get_stringal, ber_get_stringbv, ber_get_null, ber_get_boolean, ber_get_bitstring, ber_first_element, ber_next_element − OpenLDAP LBER simplified Basic Encoding Rules library routines for decoding

LIBRARY

OpenLDAP LBER (liblber, -llber)

SYNOPSIS

#include <lber.h>

ber_tag_t ber_get_next(Sockbuf ****sb*, ber_len_t ****len***, BerElement ****ber***);**

ber_tag_t ber_skip_tag(BerElement ****ber*, ber_len_t ****len***);**

ber_tag_t ber_peek_tag(BerElement ****ber*, ber_len_t ****len***);**

ber_tag_t ber_scanf(BerElement ****ber*, const char *** *fmt***, ...);**

ber_tag_t ber_get_int(BerElement ****ber*, ber_int_t ****num***);**

ber_tag_t ber_get_enum(BerElement ****ber*, ber_int_t ****num***);**

ber_tag_t ber_get_stringb(BerElement ****ber*, char ****buf***, ber_len_t ****len***);**

ber_tag_t ber_get_stringa(BerElement ****ber*, char *****buf***);**

ber_tag_t ber_get_stringal(BerElement ****ber*, struct berval *****bv***);**

ber_tag_t ber_get_stringbv(BerElement ****ber*, struct berval ****bv***, int** *alloc***);**

ber_tag_t ber_get_null(BerElement ****ber*);**

ber_tag_t ber_get_boolean(BerElement ****ber*, ber_int_t ****bool***);**

ber_tag_t ber_get_bitstringa(BerElement ****ber*, char *****buf***, ber_len_t ****blen***);**

ber_tag_t ber_first_element(BerElement ****ber*, ber_len_t ****len***, char *****cookie***);**

ber_tag_t ber_next_element(BerElement ****ber*, ber_len_t ****len***, const char ****cookie***);**

DESCRIPTION

These routines provide a subroutine interface to a simplified implementation of the Basic Encoding Rules of ASN.1. The version of BER these routines support is the one defined for the LDAP protocol. The encoding rules are the same as BER, except that only definite form lengths are used, and bitstrings and octet strings are always encoded in primitive form. This man page describes the decoding routines in the lber library. See **lber-encode**(3) for details on the corresponding encoding routines. Consult **lber-types**(3) for information about types, allocators, and deallocators.

Normally, the only routines that need to be called by an application are **ber_get_next**() to get the next BER element and **ber_scanf**() to do the actual decoding. In some cases, **ber_peek_tag**() may also need to be called in normal usage. The other routines are provided for those applications that need more control than **ber_scanf**() provides. In general, these routines return the tag of the element decoded, or LBER_ERROR if an error occurred.

The **ber_get_next**() routine is used to read the next BER element from the given Sockbuf, *sb*. It strips off and returns the leading tag, strips off and returns the length of the entire element in *len*, and sets up *ber* for subsequent calls to **ber_scanf**() et al to decode the element. See **lber-sockbuf**(3) for details of the Sockbuf implementation of the *sb* parameter.

The **ber_scanf**() routine is used to decode a BER element in much the same way that **scanf**(3) works. It reads from *ber*, a pointer to a BerElement such as returned by **ber_get_next**(), interprets the bytes according to the format string *fmt*, and stores the results in its additional arguments. The format string contains conversion specifications which are used to direct the interpretation of the BER element. The format string can contain the following characters.

a Octet string. A char ** should be supplied. Memory is allocated, filled with the contents of the octet string, null-terminated, and returned in the parameter. The caller should free the returned string using **ber_memfree**().

A Octet string. A variant of "**a**". A char ** should be supplied. Memory is allocated, filled with the contents of the octet string, null-terminated, and returned in the parameter, unless a zero-length string would result; in that case, the arg is set to NULL. The caller should free the returned string using **ber_memfree**().

s Octet string. A char * buffer should be supplied, followed by a pointer to a ber_len_t initialized to the size of the buffer. Upon return, the null-terminated octet string is put into the buffer, and the ber_len_t is set to the actual size of the octet string.

O Octet string. A struct ber_val ** should be supplied, which upon return points to a dynamically allocated struct berval containing the octet string and its length. The caller should free the returned structure using **ber_bvfree**().

o Octet string. A struct ber_val * should be supplied, which upon return contains the dynamically allocated octet string and its length. The caller should free the returned octet string using **ber_memfree**().

m Octet string. A struct ber_val * should be supplied, which upon return contains the octet string and its length. The string resides in memory assigned to the BerElement, and must not be freed by the caller.

b Boolean. A pointer to a ber_int_t should be supplied.

e Enumeration. A pointer to a ber_int_t should be supplied.

i Integer. A pointer to a ber_int_t should be supplied.

B Bitstring. A char ** should be supplied which will point to the dynamically allocated bits, followed by a ber_len_t *, which will point to the length (in bits) of the bitstring returned.

n Null. No parameter is required. The element is simply skipped if it is recognized.

v Sequence of octet strings. A char *** should be supplied, which upon return points to a dynamically allocated null-terminated array of char *'s containing the octet strings. NULL is returned if the sequence is empty. The caller should free the returned array and octet strings using **ber_memvfree**().

V Sequence of octet strings with lengths. A struct berval *** should be supplied, which upon return points to a dynamically allocated null-terminated array of struct berval *'s containing the octet strings and their lengths. NULL is returned if the sequence is empty. The caller should free the returned structures using **ber_bvecfree**().

W Sequence of octet strings with lengths. A BerVarray * should be supplied, which upon return points to a dynamically allocated array of struct berval's containing the octet strings and their lengths. The array is terminated by a struct berval with a NULL bv_val string pointer. NULL is returned if the sequence is empty. The caller should free the returned structures using **ber_bvarray_free**().

M Sequence of octet strings with lengths. This is a generalized form of the previous three formats. A void ** (ptr) should be supplied, followed by a ber_len_t * (len) and a ber_len_t (off). Upon return (ptr) will point to a dynamically allocated array whose elements are all of size (*len). A struct berval will be filled starting at offset (off) in each element. The strings in each struct berval reside in memory assigned to the BerElement and must not be freed by the caller. The array is terminated by a struct berval with a NULL bv_val string pointer. NULL is returned if the sequence is empty. The number of elements in the array is also stored in (*len) on return. The caller should free the returned array using **ber_memfree**().

l Length of the next element. A pointer to a ber_len_t should be supplied.

t Tag of the next element. A pointer to a ber_tag_t should be supplied.

T Skip element and return its tag. A pointer to a ber_tag_t should be supplied.

x Skip element. The next element is skipped.

{ Begin sequence. No parameter is required. The initial sequence tag and length are skipped.

} End sequence. No parameter is required and no action is taken.

[Begin set. No parameter is required. The initial set tag and length are skipped.

] End set. No parameter is required and no action is taken.

The **ber_get_int**() routine tries to interpret the next element as an integer, returning the result in *num*. The tag of whatever it finds is returned on success, LBER_ERROR (−1) on failure.

The **ber_get_stringb**() routine is used to read an octet string into a preallocated buffer. The *len* parameter should be initialized to the size of the buffer, and will contain the length of the octet string read upon return. The buffer should be big enough to take the octet string value plus a terminating NULL byte.

The **ber_get_stringa**() routine is used to dynamically allocate space into which an octet string is read. The caller should free the returned string using **ber_memfree().**

The **ber_get_stringal**() routine is used to dynamically allocate space into which an octet string and its length are read. It takes a struct berval **, and returns the result in this parameter. The caller should free the returned structure using **ber_bvfree().**

The **ber_get_stringbv**() routine is used to read an octet string and its length into the provided struct berval *. If the *alloc* parameter is zero, the string will reside in memory assigned to the BerElement, and must not be freed by the caller. If the *alloc* parameter is non-zero, the string will be copied into dynamically allocated space which should be returned using **ber_memfree**().

The **ber_get_null**() routine is used to read a NULL element. It returns the tag of the element it skips over.

The **ber_get_boolean**() routine is used to read a boolean value. It is called the same way that **ber_get_int**() is called.

The **ber_get_enum**() routine is used to read a enumeration value. It is called the same way that **ber_get_int**() is called.

The **ber_get_bitstringa**() routine is used to read a bitstring value. It takes a char ** which will hold the dynamically allocated bits, followed by an ber_len_t *, which will point to the length (in bits) of the bitstring returned. The caller should free the returned string using **ber_memfree**().

The **ber_first_element**() routine is used to return the tag and length of the first element in a set or sequence. It also returns in *cookie* a magic cookie parameter that should be passed to subsequent calls to ber_next_element(), which returns similar information.

EXAMPLES

Assume the variable *ber* contains a lightweight BER encoding of the following ASN.1 object:

```
AlmostASearchRequest := SEQUENCE {
   baseObject      DistinguishedName,
   scope           ENUMERATED {
      baseObject    (0),
      singleLevel   (1),
      wholeSubtree  (2)
   },
   derefAliases    ENUMERATED {
      neverDerefaliases   (0),
      derefInSearching    (1),
      derefFindingBaseObj (2),
      alwaysDerefAliases  (3)
   },
```

```
    sizelimit       INTEGER (0 .. 65535),
    timelimit       INTEGER (0 .. 65535),
    attrsOnly       BOOLEAN,
    attributes      SEQUENCE OF AttributeType
}
```

The element can be decoded using **ber_scanf**() as follows.

```
ber_int_t    scope, deref, size, time, attrsonly;
char   *dn, **attrs;
ber_tag_t tag;

tag = ber_scanf( ber, "{aeeiib{v}}",
    &dn, &scope, &deref,
    &size, &time, &attrsonly, &attrs );

if( tag == LBER_ERROR ) {
        /* error */
} else {
        /* success */
}

ber_memfree( dn );
ber_memvfree( attrs );
```

ERRORS

If an error occurs during decoding, generally these routines return LBER_ERROR ((ber_tag_t)−1).

NOTES

The return values for all of these functions are declared in the **<lber.h>** header file. Some routines may dynamically allocate memory which must be freed by the caller using supplied deallocation routines.

SEE ALSO

lber-encode(3), **lber-memory**(3), **lber-sockbuf**(3), **lber-types**(3)

ACKNOWLEDGEMENTS

NAME

ber_get_next, ber_skip_tag, ber_peek_tag, ber_scanf, ber_get_int, ber_get_enum, ber_get_stringb, ber_get_stringa, ber_get_stringal, ber_get_stringbv, ber_get_null, ber_get_boolean, ber_get_bitstring, ber_first_element, ber_next_element − OpenLDAP LBER simplified Basic Encoding Rules library routines for decoding

LIBRARY

OpenLDAP LBER (liblber, -llber)

SYNOPSIS

#include <lber.h>

ber_tag_t ber_get_next(Sockbuf ****sb*, ber_len_t ****len***, BerElement ****ber***);**

ber_tag_t ber_skip_tag(BerElement ****ber*, ber_len_t ****len***);**

ber_tag_t ber_peek_tag(BerElement ****ber*, ber_len_t ****len***);**

ber_tag_t ber_scanf(BerElement ****ber*, const char *** *fmt***, ...);**

ber_tag_t ber_get_int(BerElement ****ber*, ber_int_t ****num***);**

ber_tag_t ber_get_enum(BerElement ****ber*, ber_int_t ****num***);**

ber_tag_t ber_get_stringb(BerElement ****ber*, char ****buf***, ber_len_t ****len***);**

ber_tag_t ber_get_stringa(BerElement ****ber*, char *****buf***);**

ber_tag_t ber_get_stringal(BerElement ****ber*, struct berval *****bv***);**

ber_tag_t ber_get_stringbv(BerElement ****ber*, struct berval ****bv***, int** *alloc***);**

ber_tag_t ber_get_null(BerElement ****ber*);**

ber_tag_t ber_get_boolean(BerElement ****ber*, ber_int_t ****bool***);**

ber_tag_t ber_get_bitstringa(BerElement ****ber*, char *****buf***, ber_len_t ****blen***);**

ber_tag_t ber_first_element(BerElement ****ber*, ber_len_t ****len***, char *****cookie***);**

ber_tag_t ber_next_element(BerElement ****ber*, ber_len_t ****len***, const char ****cookie***);**

DESCRIPTION

These routines provide a subroutine interface to a simplified implementation of the Basic Encoding Rules of ASN.1. The version of BER these routines support is the one defined for the LDAP protocol. The encoding rules are the same as BER, except that only definite form lengths are used, and bitstrings and octet strings are always encoded in primitive form. This man page describes the decoding routines in the lber library. See **lber-encode**(3) for details on the corresponding encoding routines. Consult **lber-types**(3) for information about types, allocators, and deallocators.

Normally, the only routines that need to be called by an application are **ber_get_next**() to get the next BER element and **ber_scanf**() to do the actual decoding. In some cases, **ber_peek_tag**() may also need to be called in normal usage. The other routines are provided for those applications that need more control than **ber_scanf**() provides. In general, these routines return the tag of the element decoded, or LBER_ERROR if an error occurred.

The **ber_get_next**() routine is used to read the next BER element from the given Sockbuf, *sb*. It strips off and returns the leading tag, strips off and returns the length of the entire element in *len*, and sets up *ber* for subsequent calls to **ber_scanf**() et al to decode the element. See **lber-sockbuf**(3) for details of the Sockbuf implementation of the *sb* parameter.

The **ber_scanf**() routine is used to decode a BER element in much the same way that **scanf**(3) works. It reads from *ber*, a pointer to a BerElement such as returned by **ber_get_next**(), interprets the bytes according to the format string *fmt*, and stores the results in its additional arguments. The format string contains conversion specifications which are used to direct the interpretation of the BER element. The format string can contain the following characters.

a Octet string. A char ** should be supplied. Memory is allocated, filled with the contents of the octet string, null-terminated, and returned in the parameter. The caller should free the returned string using **ber_memfree**().

A Octet string. A variant of "**a**". A char ** should be supplied. Memory is allocated, filled with the contents of the octet string, null-terminated, and returned in the parameter, unless a zero-length string would result; in that case, the arg is set to NULL. The caller should free the returned string using **ber_memfree**().

s Octet string. A char * buffer should be supplied, followed by a pointer to a ber_len_t initialized to the size of the buffer. Upon return, the null-terminated octet string is put into the buffer, and the ber_len_t is set to the actual size of the octet string.

O Octet string. A struct ber_val ** should be supplied, which upon return points to a dynamically allocated struct berval containing the octet string and its length. The caller should free the returned structure using **ber_bvfree**().

o Octet string. A struct ber_val * should be supplied, which upon return contains the dynamically allocated octet string and its length. The caller should free the returned octet string using **ber_memfree**().

m Octet string. A struct ber_val * should be supplied, which upon return contains the octet string and its length. The string resides in memory assigned to the BerElement, and must not be freed by the caller.

b Boolean. A pointer to a ber_int_t should be supplied.

e Enumeration. A pointer to a ber_int_t should be supplied.

i Integer. A pointer to a ber_int_t should be supplied.

B Bitstring. A char ** should be supplied which will point to the dynamically allocated bits, followed by a ber_len_t *, which will point to the length (in bits) of the bitstring returned.

n Null. No parameter is required. The element is simply skipped if it is recognized.

v Sequence of octet strings. A char *** should be supplied, which upon return points to a dynamically allocated null-terminated array of char *'s containing the octet strings. NULL is returned if the sequence is empty. The caller should free the returned array and octet strings using **ber_memvfree**().

V Sequence of octet strings with lengths. A struct berval *** should be supplied, which upon return points to a dynamically allocated null-terminated array of struct berval *'s containing the octet strings and their lengths. NULL is returned if the sequence is empty. The caller should free the returned structures using **ber_bvecfree**().

W Sequence of octet strings with lengths. A BerVarray * should be supplied, which upon return points to a dynamically allocated array of struct berval's containing the octet strings and their lengths. The array is terminated by a struct berval with a NULL bv_val string pointer. NULL is returned if the sequence is empty. The caller should free the returned structures using **ber_bvarray_free**().

M Sequence of octet strings with lengths. This is a generalized form of the previous three formats. A void ** (ptr) should be supplied, followed by a ber_len_t * (len) and a ber_len_t (off). Upon return (ptr) will point to a dynamically allocated array whose elements are all of size (*len). A struct berval will be filled starting at offset (off) in each element. The strings in each struct berval reside in memory assigned to the BerElement and must not be freed by the caller. The array is terminated by a struct berval with a NULL bv_val string pointer. NULL is returned if the sequence is empty. The number of elements in the array is also stored in (*len) on return. The caller should free the returned array using **ber_memfree**().

l Length of the next element. A pointer to a ber_len_t should be supplied.

t Tag of the next element. A pointer to a ber_tag_t should be supplied.

T Skip element and return its tag. A pointer to a ber_tag_t should be supplied.

x Skip element. The next element is skipped.

{ Begin sequence. No parameter is required. The initial sequence tag and length are skipped.

} End sequence. No parameter is required and no action is taken.

[Begin set. No parameter is required. The initial set tag and length are skipped.

] End set. No parameter is required and no action is taken.

The **ber_get_int**() routine tries to interpret the next element as an integer, returning the result in *num*. The tag of whatever it finds is returned on success, LBER_ERROR (−1) on failure.

The **ber_get_stringb**() routine is used to read an octet string into a preallocated buffer. The *len* parameter should be initialized to the size of the buffer, and will contain the length of the octet string read upon return. The buffer should be big enough to take the octet string value plus a terminating NULL byte.

The **ber_get_stringa**() routine is used to dynamically allocate space into which an octet string is read. The caller should free the returned string using **ber_memfree().**

The **ber_get_stringal**() routine is used to dynamically allocate space into which an octet string and its length are read. It takes a struct berval **, and returns the result in this parameter. The caller should free the returned structure using **ber_bvfree().**

The **ber_get_stringbv**() routine is used to read an octet string and its length into the provided struct berval *. If the *alloc* parameter is zero, the string will reside in memory assigned to the BerElement, and must not be freed by the caller. If the *alloc* parameter is non-zero, the string will be copied into dynamically allocated space which should be returned using **ber_memfree**().

The **ber_get_null**() routine is used to read a NULL element. It returns the tag of the element it skips over.

The **ber_get_boolean**() routine is used to read a boolean value. It is called the same way that **ber_get_int**() is called.

The **ber_get_enum**() routine is used to read a enumeration value. It is called the same way that **ber_get_int**() is called.

The **ber_get_bitstringa**() routine is used to read a bitstring value. It takes a char ** which will hold the dynamically allocated bits, followed by an ber_len_t *, which will point to the length (in bits) of the bitstring returned. The caller should free the returned string using **ber_memfree**().

The **ber_first_element**() routine is used to return the tag and length of the first element in a set or sequence. It also returns in *cookie* a magic cookie parameter that should be passed to subsequent calls to ber_next_element(), which returns similar information.

EXAMPLES

Assume the variable *ber* contains a lightweight BER encoding of the following ASN.1 object:

```
AlmostASearchRequest := SEQUENCE {
   baseObject      DistinguishedName,
   scope           ENUMERATED {
      baseObject    (0),
      singleLevel   (1),
      wholeSubtree  (2)
   },
   derefAliases    ENUMERATED {
      neverDerefaliases   (0),
      derefInSearching    (1),
      derefFindingBaseObj (2),
      alwaysDerefAliases  (3)
   },
```

```
    sizelimit       INTEGER (0 .. 65535),
    timelimit       INTEGER (0 .. 65535),
    attrsOnly       BOOLEAN,
    attributes      SEQUENCE OF AttributeType
}
```

The element can be decoded using **ber_scanf**() as follows.

```
ber_int_t    scope, deref, size, time, attrsonly;
char   *dn, **attrs;
ber_tag_t tag;

tag = ber_scanf( ber, "{aeeiib{v}}",
   &dn, &scope, &deref,
   &size, &time, &attrsonly, &attrs );

if( tag == LBER_ERROR ) {
      /* error */
} else {
      /* success */
}

ber_memfree( dn );
ber_memvfree( attrs );
```

ERRORS

If an error occurs during decoding, generally these routines return LBER_ERROR ((ber_tag_t)−1).

NOTES

The return values for all of these functions are declared in the **<lber.h>** header file. Some routines may dynamically allocate memory which must be freed by the caller using supplied deallocation routines.

SEE ALSO

lber-encode(3), **lber-memory**(3), **lber-sockbuf**(3), **lber-types**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ber_alloc_t, ber_flush, ber_flush2, ber_printf, ber_put_int, ber_put_enum, ber_put_ostring, ber_put_string, ber_put_null, ber_put_boolean, ber_put_bitstring, ber_start_seq, ber_start_set, ber_put_seq, ber_put_set − OpenLDAP LBER simplified Basic Encoding Rules library routines for encoding

LIBRARY

OpenLDAP LBER (liblber, -llber)

SYNOPSIS

#include <lber.h>

BerElement *ber_alloc_t(int *options***);**

int ber_flush(Sockbuf ***sb*, BerElement ***ber***, int** *freeit***);**

int ber_flush2(Sockbuf ***sb*, BerElement ***ber***, int** *freeit***);**

int ber_printf(BerElement ***ber*, const char *** *fmt***, ...);**

int ber_put_int(BerElement ***ber*, ber_int_t** *num***, ber_tag_t** *tag***);**

int ber_put_enum(BerElement ***ber*, ber_int_t** *num***, ber_tag_t** *tag***);**

int ber_put_ostring(BerElement ***ber*, const char ***str***, ber_len_t** *len***, ber_tag_t** *tag***);**

int ber_put_string(BerElement ***ber*, const char ***str***, ber_tag_t** *tag***);**

int ber_put_null(BerElement ***ber*, ber_tag_t** *tag***);**

int ber_put_boolean(BerElement ***ber*, ber_int_t** *bool***, ber_tag_t** *tag***);**

int ber_put_bitstring(BerElement ***ber*, const char ***str***, ber_len_t** *blen***, ber_tag_t** *tag***);**

int ber_start_seq(BerElement ***ber*, ber_tag_t** *tag***);**

int ber_start_set(BerElement ***ber*, ber_tag_t** *tag***);**

int ber_put_seq(BerElement ***ber*);**

int ber_put_set(BerElement ***ber*);**

DESCRIPTION

These routines provide a subroutine interface to a simplified implementation of the Basic Encoding Rules of ASN.1. The version of BER these routines support is the one defined for the LDAP protocol. The encoding rules are the same as BER, except that only definite form lengths are used, and bitstrings and octet strings are always encoded in primitive form. This man page describes the encoding routines in the lber library. See **lber-decode**(3) for details on the corresponding decoding routines. Consult **lber-types**(3) for information about types, allocators, and deallocators.

Normally, the only routines that need to be called by an application are **ber_alloc_t**() to allocate a BER element for encoding, **ber_printf**() to do the actual encoding, and **ber_flush2**() to actually write the element. The other routines are provided for those applications that need more control than **ber_printf**() provides. In general, these routines return the length of the element encoded, or -1 if an error occurred.

The **ber_alloc_t**() routine is used to allocate a new BER element. It should be called with an argument of LBER_USE_DER.

The **ber_flush2**() routine is used to actually write the element to a socket (or file) descriptor, once it has been fully encoded (using **ber_printf**() and friends). See **lber-sockbuf**(3) for more details on the Sockbuf implementation of the *sb* parameter. If the *freeit* parameter is non-zero, the supplied *ber* will be freed. If *LBER_FLUSH_FREE_ON_SUCCESS* is used, the *ber* is only freed when successfully flushed, otherwise it is left intact; if *LBER_FLUSH_FREE_ON_ERROR* is used, the *ber* is only freed when an error occurs, otherwise it is left intact; if *LBER_FLUSH_FREE_ALWAYS* is used, the *ber* is freed anyway. This function differs from the original **ber_flush**(3) function, whose behavior corresponds to that indicated for *LBER_FLUSH_FREE_ON_SUCCESS*. Note that in the future, the behavior of **ber_flush**(3) with *freeit* non-zero might change into that of **ber_flush2**(3) with *freeit* set to *LBER_FLUSH_FREE_ALWAYS*.

The **ber_printf**() routine is used to encode a BER element in much the same way that **sprintf**(3) works.

One important difference, though, is that some state information is kept with the *ber* parameter so that multiple calls can be made to **ber_printf**() to append things to the end of the BER element. **Ber_printf**() writes to *ber*, a pointer to a BerElement such as returned by **ber_alloc_t**(). It interprets and formats its arguments according to the format string *fmt*. The format string can contain the following characters:

- **b** Boolean. An ber_int_t parameter should be supplied. A boolean element is output.
- **e** Enumeration. An ber_int_t parameter should be supplied. An enumeration element is output.
- **i** Integer. An ber_int_t parameter should be supplied. An integer element is output.
- **B** Bitstring. A char * pointer to the start of the bitstring is supplied, followed by the number of bits in the bitstring. A bitstring element is output.
- **n** Null. No parameter is required. A null element is output.
- **o** Octet string. A char * is supplied, followed by the length of the string pointed to. An octet string element is output.
- **O** Octet string. A struct berval * is supplied. An octet string element is output.
- **s** Octet string. A null-terminated string is supplied. An octet string element is output, not including the trailing NULL octet.
- **t** Tag. A ber_tag_t specifying the tag to give the next element is provided. This works across calls.
- **v** Several octet strings. A null-terminated array of char *'s is supplied. Note that a construct like '{v}' is required to get an actual SEQUENCE OF octet strings.
- **V** Several octet strings. A null-terminated array of struct berval *'s is supplied. Note that a construct like '{V}' is required to get an actual SEQUENCE OF octet strings.
- **W** Several octet strings. An array of struct berval's is supplied. The array is terminated by a struct berval with a NULL bv_val. Note that a construct like '{W}' is required to get an actual SEQUENCE OF octet strings.
- **{** Begin sequence. No parameter is required.
- **}** End sequence. No parameter is required.
- **[** Begin set. No parameter is required.
- **]** End set. No parameter is required.

The **ber_put_int**() routine writes the integer element *num* to the BER element *ber*.

The **ber_put_enum**() routine writes the enumeration element *num* to the BER element *ber*.

The **ber_put_boolean**() routine writes the boolean value given by *bool* to the BER element.

The **ber_put_bitstring**() routine writes *blen* bits starting at *str* as a bitstring value to the given BER element. Note that *blen* is the length *in bits* of the bitstring.

The **ber_put_ostring**() routine writes *len* bytes starting at *str* to the BER element as an octet string.

The **ber_put_string**() routine writes the null-terminated string (minus the terminating ' ') to the BER element as an octet string.

The **ber_put_null**() routine writes a NULL element to the BER element.

The **ber_start_seq**() routine is used to start a sequence in the BER element. The **ber_start_set**() routine works similarly. The end of the sequence or set is marked by the nearest matching call to **ber_put_seq**() or **ber_put_set**(), respectively.

EXAMPLES

Assuming the following variable declarations, and that the variables have been assigned appropriately, an lber encoding of the following ASN.1 object:

```
AlmostASearchRequest := SEQUENCE {
```

```
        baseObject      DistinguishedName,
        scope           ENUMERATED {
            baseObject    (0),
            singleLevel   (1),
            wholeSubtree  (2)
        },
        derefAliases    ENUMERATED {
            neverDerefaliases   (0),
            derefInSearching    (1),
            derefFindingBaseObj (2),
            alwaysDerefAliases  (3)
        },
        sizelimit       INTEGER (0 .. 65535),
        timelimit       INTEGER (0 .. 65535),
        attrsOnly       BOOLEAN,
        attributes      SEQUENCE OF AttributeType
    }
```

can be achieved like so:

```
    int rc;
    ber_int_t    scope, ali, size, time, attrsonly;
    char   *dn, **attrs;
    BerElement *ber;

    /* ... fill in values ... */

    ber = ber_alloc_t( LBER_USE_DER );

    if ( ber == NULL ) {
        /* error */
    }

    rc = ber_printf( ber, "{siiiib{v}}", dn, scope, ali,
        size, time, attrsonly, attrs );

    if( rc == -1 ) {
        /* error */
    } else {
        /* success */
    }
```

ERRORS

If an error occurs during encoding, generally these routines return -1.

NOTES

The return values for all of these functions are declared in the <lber.h> header file.

SEE ALSO

lber-decode(3), **lber-memory**(3), **lber-sockbuf**(3), **lber-types**(3)

ACKNOWLEDGEMENTS

NAME

ber_alloc_t, ber_flush, ber_flush2, ber_printf, ber_put_int, ber_put_enum, ber_put_ostring, ber_put_string, ber_put_null, ber_put_boolean, ber_put_bitstring, ber_start_seq, ber_start_set, ber_put_seq, ber_put_set − OpenLDAP LBER simplified Basic Encoding Rules library routines for encoding

LIBRARY

OpenLDAP LBER (liblber, -llber)

SYNOPSIS

#include <lber.h>

BerElement *ber_alloc_t(int *options***);**

int ber_flush(Sockbuf ***sb*, BerElement ***ber***, int** *freeit***);**

int ber_flush2(Sockbuf ***sb*, BerElement ***ber***, int** *freeit***);**

int ber_printf(BerElement ***ber*, const char *** *fmt***, ...);**

int ber_put_int(BerElement ***ber*, ber_int_t** *num***, ber_tag_t** *tag***);**

int ber_put_enum(BerElement ***ber*, ber_int_t** *num***, ber_tag_t** *tag***);**

int ber_put_ostring(BerElement ***ber*, const char ***str***, ber_len_t** *len***, ber_tag_t** *tag***);**

int ber_put_string(BerElement ***ber*, const char ***str***, ber_tag_t** *tag***);**

int ber_put_null(BerElement ***ber*, ber_tag_t** *tag***);**

int ber_put_boolean(BerElement ***ber*, ber_int_t** *bool***, ber_tag_t** *tag***);**

int ber_put_bitstring(BerElement ***ber*, const char ***str***, ber_len_t** *blen***, ber_tag_t** *tag***);**

int ber_start_seq(BerElement ***ber*, ber_tag_t** *tag***);**

int ber_start_set(BerElement ***ber*, ber_tag_t** *tag***);**

int ber_put_seq(BerElement ***ber*);**

int ber_put_set(BerElement ***ber*);**

DESCRIPTION

These routines provide a subroutine interface to a simplified implementation of the Basic Encoding Rules of ASN.1. The version of BER these routines support is the one defined for the LDAP protocol. The encoding rules are the same as BER, except that only definite form lengths are used, and bitstrings and octet strings are always encoded in primitive form. This man page describes the encoding routines in the lber library. See **lber-decode**(3) for details on the corresponding decoding routines. Consult **lber-types**(3) for information about types, allocators, and deallocators.

Normally, the only routines that need to be called by an application are **ber_alloc_t**() to allocate a BER element for encoding, **ber_printf**() to do the actual encoding, and **ber_flush2**() to actually write the element. The other routines are provided for those applications that need more control than **ber_printf**() provides. In general, these routines return the length of the element encoded, or -1 if an error occurred.

The **ber_alloc_t**() routine is used to allocate a new BER element. It should be called with an argument of LBER_USE_DER.

The **ber_flush2**() routine is used to actually write the element to a socket (or file) descriptor, once it has been fully encoded (using **ber_printf**() and friends). See **lber-sockbuf**(3) for more details on the Sockbuf implementation of the *sb* parameter. If the *freeit* parameter is non-zero, the supplied *ber* will be freed. If *LBER_FLUSH_FREE_ON_SUCCESS* is used, the *ber* is only freed when successfully flushed, otherwise it is left intact; if *LBER_FLUSH_FREE_ON_ERROR* is used, the *ber* is only freed when an error occurs, otherwise it is left intact; if *LBER_FLUSH_FREE_ALWAYS* is used, the *ber* is freed anyway. This function differs from the original **ber_flush**(3) function, whose behavior corresponds to that indicated for *LBER_FLUSH_FREE_ON_SUCCESS*. Note that in the future, the behavior of **ber_flush**(3) with *freeit* non-zero might change into that of **ber_flush2**(3) with *freeit* set to *LBER_FLUSH_FREE_ALWAYS*.

The **ber_printf**() routine is used to encode a BER element in much the same way that **sprintf**(3) works.

One important difference, though, is that some state information is kept with the *ber* parameter so that multiple calls can be made to **ber_printf**() to append things to the end of the BER element. **Ber_printf**() writes to *ber*, a pointer to a BerElement such as returned by **ber_alloc_t**(). It interprets and formats its arguments according to the format string *fmt*. The format string can contain the following characters:

b Boolean. An ber_int_t parameter should be supplied. A boolean element is output.

e Enumeration. An ber_int_t parameter should be supplied. An enumeration element is output.

i Integer. An ber_int_t parameter should be supplied. An integer element is output.

B Bitstring. A char * pointer to the start of the bitstring is supplied, followed by the number of bits in the bitstring. A bitstring element is output.

n Null. No parameter is required. A null element is output.

o Octet string. A char * is supplied, followed by the length of the string pointed to. An octet string element is output.

O Octet string. A struct berval * is supplied. An octet string element is output.

s Octet string. A null-terminated string is supplied. An octet string element is output, not including the trailing NULL octet.

t Tag. A ber_tag_t specifying the tag to give the next element is provided. This works across calls.

v Several octet strings. A null-terminated array of char *'s is supplied. Note that a construct like '{v}' is required to get an actual SEQUENCE OF octet strings.

V Several octet strings. A null-terminated array of struct berval *'s is supplied. Note that a construct like '{V}' is required to get an actual SEQUENCE OF octet strings.

W Several octet strings. An array of struct berval's is supplied. The array is terminated by a struct berval with a NULL bv_val. Note that a construct like '{W}' is required to get an actual SEQUENCE OF octet strings.

{ Begin sequence. No parameter is required.

} End sequence. No parameter is required.

[Begin set. No parameter is required.

] End set. No parameter is required.

The **ber_put_int**() routine writes the integer element *num* to the BER element *ber*.

The **ber_put_enum**() routine writes the enumeration element *num* to the BER element *ber*.

The **ber_put_boolean**() routine writes the boolean value given by *bool* to the BER element.

The **ber_put_bitstring**() routine writes *blen* bits starting at *str* as a bitstring value to the given BER element. Note that *blen* is the length *in bits* of the bitstring.

The **ber_put_ostring**() routine writes *len* bytes starting at *str* to the BER element as an octet string.

The **ber_put_string**() routine writes the null-terminated string (minus the terminating ' ') to the BER element as an octet string.

The **ber_put_null**() routine writes a NULL element to the BER element.

The **ber_start_seq**() routine is used to start a sequence in the BER element. The **ber_start_set**() routine works similarly. The end of the sequence or set is marked by the nearest matching call to **ber_put_seq**() or **ber_put_set**(), respectively.

EXAMPLES

Assuming the following variable declarations, and that the variables have been assigned appropriately, an lber encoding of the following ASN.1 object:

```
AlmostASearchRequest := SEQUENCE {
```

```
    baseObject      DistinguishedName,
    scope           ENUMERATED {
        baseObject    (0),
        singleLevel   (1),
        wholeSubtree  (2)
    },
    derefAliases    ENUMERATED {
        neverDerefaliases   (0),
        derefInSearching    (1),
        derefFindingBaseObj (2),
        alwaysDerefAliases  (3)
    },
    sizelimit       INTEGER (0 .. 65535),
    timelimit       INTEGER (0 .. 65535),
    attrsOnly       BOOLEAN,
    attributes      SEQUENCE OF AttributeType
}
```

can be achieved like so:

```
int rc;
ber_int_t    scope, ali, size, time, attrsonly;
char   *dn, **attrs;
BerElement *ber;

/* ... fill in values ... */

ber = ber_alloc_t( LBER_USE_DER );

if ( ber == NULL ) {
        /* error */
}

rc = ber_printf( ber, "{siiiib{v}}", dn, scope, ali,
    size, time, attrsonly, attrs );

if( rc == -1 ) {
        /* error */
} else {
        /* success */
}
```

ERRORS

If an error occurs during encoding, generally these routines return -1.

NOTES

The return values for all of these functions are declared in the <lber.h> header file.

SEE ALSO

lber-decode(3), **lber-memory**(3), **lber-sockbuf**(3), **lber-types**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ber_memalloc, ber_memcalloc, ber_memrealloc, ber_memfree, ber_memvfree − OpenLDAP LBER memory allocators

LIBRARY

OpenLDAP LBER (liblber, -llber)

SYNOPSIS

#include <lber.h>

void *ber_memalloc(ber_len_t *bytes***);**

void *ber_memcalloc(ber_len_t *nelems***, ber_len_t** *bytes***);**

void *ber_memrealloc(void * *ptr***, ber_len_t** *bytes***);**

void ber_memfree(void * *ptr***);**

void ber_memvfree(void ** *vec***);**

DESCRIPTION

These routines are used to allocate/deallocate memory used/returned by the Lightweight BER library as required by **lber-encode**(3) and **lber-decode**(3). **ber_memalloc**(), **ber_memcalloc**(), **ber_memrealloc**(), and **ber_memfree**() are used exactly like the standard **malloc**(3), **calloc**(3), **realloc**(3), and **free**(3) routines, respectively. The **ber_memvfree**() routine is used to free a dynamically allocated array of pointers to arbitrary dynamically allocated objects.

SEE ALSO

lber-decode(3), **lber-encode**(3), **lber-types**(3)

ACKNOWLEDGEMENTS

NAME

ber_memalloc, ber_memcalloc, ber_memrealloc, ber_memfree, ber_memvfree − OpenLDAP LBER memory allocators

LIBRARY

OpenLDAP LBER (liblber, -llber)

SYNOPSIS

#include <lber.h>

void *ber_memalloc(ber_len_t *bytes***);**

void *ber_memcalloc(ber_len_t *nelems***, ber_len_t** *bytes***);**

void *ber_memrealloc(void * *ptr***, ber_len_t** *bytes***);**

void ber_memfree(void * *ptr***);**

void ber_memvfree(void ** *vec***);**

DESCRIPTION

These routines are used to allocate/deallocate memory used/returned by the Lightweight BER library as required by **lber-encode**(3) and **lber-decode**(3). **ber_memalloc**(), **ber_memcalloc**(), **ber_memrealloc**(), and **ber_memfree**() are used exactly like the standard **malloc**(3), **calloc**(3), **realloc**(3), and **free**(3) routines, respectively. The **ber_memvfree**() routine is used to free a dynamically allocated array of pointers to arbitrary dynamically allocated objects.

SEE ALSO

lber-decode(3), **lber-encode**(3), **lber-types**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ber_sockbuf_alloc, ber_sockbuf_free, ber_sockbuf_ctrl, ber_sockbuf_add_io, ber_sockbuf_remove_io, Sockbuf_IO − OpenLDAP LBER I/O infrastructure

LIBRARY

OpenLDAP LBER (liblber, -llber)

SYNOPSIS

#include <lber.h>

Sockbuf *ber_sockbuf_alloc(void);

void ber_sockbuf_free(Sockbuf * *sb* **);**

int ber_sockbuf_ctrl(Sockbuf * *sb*, **int** *opt*, **void *** *arg* **);**

int ber_sockbuf_add_io(Sockbuf * *sb*, **Sockbuf_IO *** *sbio*, **int** *layer*, **void *** *arg* **);**

int ber_sockbuf_remove_io(Sockbuf * *sb*, **Sockbuf_IO *** *sbio*, **int** *layer* **);**

typedef struct sockbuf_io_desc {
int *sbiod_level*;
Sockbuf * *sbiod_sb*;
Sockbuf_IO * *sbiod_io*;
void * *sbiod_pvt*;
struct sockbuf_io_desc * *sbiod_next*;
} Sockbuf_IO_Desc;

typedef struct sockbuf_io {
int (* *sbi_setup* **)(Sockbuf_IO_Desc *** *sbiod*, **void *** *arg* **);**
int (* *sbi_remove* **)(Sockbuf_IO_Desc *** *sbiod* **);**
int (* *sbi_ctrl* **)(Sockbuf_IO_Desc *** *sbiod*, **int** *opt*, **void *** *arg* **);**
ber_slen_t (* *sbi_read* **)(Sockbuf_IO_Desc *** *sbiod*, **void *** *buf*, **ber_len_t** *len* **);**
ber_slen_t (* *sbi_write* **)(Sockbuf_IO_Desc *** *sbiod*, **void *** *buf*, **ber_len_t** *len* **);**
int (* *sbi_close* **)(Sockbuf_IO_Desc *** *sbiod* **);**
} Sockbuf_IO;

DESCRIPTION

These routines are used to manage the low level I/O operations performed by the Lightweight BER library. They are called implicitly by the other libraries and usually do not need to be called directly from applications. The I/O framework is modularized and new transport layers can be supported by appropriately defining a **Sockbuf_IO** structure and installing it onto an existing **Sockbuf**. **Sockbuf** structures are allocated and freed by **ber_sockbuf_alloc**() and **ber_sockbuf_free**(), respectively. The **ber_sockbuf_ctrl**() function is used to get and set options related to a **Sockbuf** or to a specific I/O layer of the **Sockbuf**. The **ber_sockbuf_add_io**() and **ber_sockbuf_remove_io**() functions are used to add and remove specific I/O layers on a **Sockbuf**.

Options for **ber_sockbuf_ctrl**() include:

LBER_SB_OPT_HAS_IO
: Takes a **Sockbuf_IO *** argument and returns 1 if the given handler is installed on the **Sockbuf**, otherwise returns 0.

LBER_SB_OPT_GET_FD
: Retrieves the file descriptor associated to the **Sockbuf**; **arg** must be a **ber_socket_t ***. The return value will be 1 if a valid descriptor was present, -1 otherwise.

LBER_SB_OPT_SET_FD
: Sets the file descriptor of the **Sockbuf** to the descriptor pointed to by **arg**; **arg** must be a **ber_socket_t ***. The return value will always be 1.

LBER_SB_OPT_SET_NONBLOCK
Toggles the non-blocking state of the file descriptor associated to the **Sockbuf**. **arg** should be NULL to disable and non-NULL to enable the non-blocking state. The return value will be 1 for success, -1 otherwise.

LBER_SB_OPT_DRAIN
Flush (read and discard) all available input on the **Sockbuf**. The return value will be 1.

LBER_SB_OPT_NEEDS_READ
Returns non-zero if input is waiting to be read.

LBER_SB_OPT_NEEDS_WRITE
Returns non-zero if the **Sockbuf** is ready to be written.

LBER_SB_OPT_GET_MAX_INCOMING
Returns the maximum allowed size of an incoming message; **arg** must be a **ber_len_t *.** The return value will be 1.

LBER_SB_OPT_SET_MAX_INCOMING
Sets the maximum allowed size of an incoming message; **arg** must be a **ber_len_t *.** The return value will be 1.

Options not in this list will be passed down to each **Sockbuf_IO** handler in turn until one of them processes it. If the option is not handled **ber_sockbuf_ctrl**() will return 0.

Multiple **Sockbuf_IO** handlers can be stacked in multiple layers to provide various functionality. Currently defined layers include

LBER_SBIOD_LEVEL_PROVIDER
the lowest layer, talking directly to a network

LBER_SBIOD_LEVEL_TRANSPORT
an intermediate layer

LBER_SBIOD_LEVEL_APPLICATION
a higher layer

Currently defined **Sockbuf_IO** handlers in liblber include

ber_sockbuf_io_tcp
The default stream-oriented provider

ber_sockbuf_io_fd
A stream-oriented provider for local IPC sockets

ber_sockbuf_io_dgram
A datagram-oriented provider. This handler is only present if the liblber library was built with LDAP_CONNECTIONLESS defined.

ber_sockbuf_io_readahead
A buffering layer, usually used with a datagram provider to hide the datagram semantics from upper layers.

ber_sockbuf_io_debug
A generic handler that outputs hex dumps of all traffic. This handler may be inserted multiple times at arbitrary layers to show the flow of data between other handlers.

Additional handlers may be present in libldap if support for them was enabled:

ldap_pvt_sockbuf_io_sasl
An application layer handler for SASL encoding/decoding.

sb_tls_sbio
A transport layer handler for SSL/TLS encoding/decoding. Note that this handler is private to the library and is not exposed in the API.

The provided handlers are all instantiated implicitly by libldap, and applications generally will not need to directly manipulate them.

SEE ALSO

lber-decode(3), **lber-encode**(3), **lber-types**(3), **ldap_get_option**(3)

ACKNOWLEDGEMENTS

NAME

ber_sockbuf_alloc, ber_sockbuf_free, ber_sockbuf_ctrl, ber_sockbuf_add_io, ber_sockbuf_remove_io, Sockbuf_IO − OpenLDAP LBER I/O infrastructure

LIBRARY

OpenLDAP LBER (liblber, -llber)

SYNOPSIS

#include <lber.h>

Sockbuf *ber_sockbuf_alloc(void);

void ber_sockbuf_free(Sockbuf * *sb* **);**

int ber_sockbuf_ctrl(Sockbuf * *sb***, int** *opt***, void *** *arg***);**

int ber_sockbuf_add_io(Sockbuf * *sb***, Sockbuf_IO *** *sbio***, int** *layer***, void *** *arg***);**

int ber_sockbuf_remove_io(Sockbuf * *sb***, Sockbuf_IO *** *sbio***, int** *layer***);**

typedef struct sockbuf_io_desc {
int *sbiod_level***;**
Sockbuf * *sbiod_sb***;**
Sockbuf_IO * *sbiod_io***;**
void * *sbiod_pvt***;**
struct sockbuf_io_desc * *sbiod_next***;**
} Sockbuf_IO_Desc;

typedef struct sockbuf_io {
int (* *sbi_setup***)(Sockbuf_IO_Desc *** *sbiod***, void *** *arg***);**
int (* *sbi_remove***)(Sockbuf_IO_Desc *** *sbiod***);**
int (* *sbi_ctrl***)(Sockbuf_IO_Desc *** *sbiod***, int** *opt***, void *** *arg***);**
ber_slen_t (* *sbi_read***)(Sockbuf_IO_Desc *** *sbiod***, void *** *buf***, ber_len_t** *len***);**
ber_slen_t (* *sbi_write***)(Sockbuf_IO_Desc *** *sbiod***, void *** *buf***, ber_len_t** *len***);**
int (* *sbi_close***)(Sockbuf_IO_Desc *** *sbiod***);**
} Sockbuf_IO;

DESCRIPTION

These routines are used to manage the low level I/O operations performed by the Lightweight BER library. They are called implicitly by the other libraries and usually do not need to be called directly from applications. The I/O framework is modularized and new transport layers can be supported by appropriately defining a **Sockbuf_IO** structure and installing it onto an existing **Sockbuf**. **Sockbuf** structures are allocated and freed by **ber_sockbuf_alloc**() and **ber_sockbuf_free**(), respectively. The **ber_sockbuf_ctrl**() function is used to get and set options related to a **Sockbuf** or to a specific I/O layer of the **Sockbuf**. The **ber_sockbuf_add_io**() and **ber_sockbuf_remove_io**() functions are used to add and remove specific I/O layers on a **Sockbuf**.

Options for **ber_sockbuf_ctrl**() include:

LBER_SB_OPT_HAS_IO

Takes a **Sockbuf_IO *** argument and returns 1 if the given handler is installed on the **Sockbuf**, otherwise returns 0.

LBER_SB_OPT_GET_FD

Retrieves the file descriptor associated to the **Sockbuf**; **arg** must be a **ber_socket_t ***. The return value will be 1 if a valid descriptor was present, -1 otherwise.

LBER_SB_OPT_SET_FD

Sets the file descriptor of the **Sockbuf** to the descriptor pointed to by **arg**; **arg** must be a **ber_socket_t ***. The return value will always be 1.

LBER_SB_OPT_SET_NONBLOCK
Toggles the non-blocking state of the file descriptor associated to the **Sockbuf**. **arg** should be NULL to disable and non-NULL to enable the non-blocking state. The return value will be 1 for success, -1 otherwise.

LBER_SB_OPT_DRAIN
Flush (read and discard) all available input on the **Sockbuf**. The return value will be 1.

LBER_SB_OPT_NEEDS_READ
Returns non-zero if input is waiting to be read.

LBER_SB_OPT_NEEDS_WRITE
Returns non-zero if the **Sockbuf** is ready to be written.

LBER_SB_OPT_GET_MAX_INCOMING
Returns the maximum allowed size of an incoming message; **arg** must be a **ber_len_t *.** The return value will be 1.

LBER_SB_OPT_SET_MAX_INCOMING
Sets the maximum allowed size of an incoming message; **arg** must be a **ber_len_t *.** The return value will be 1.

Options not in this list will be passed down to each **Sockbuf_IO** handler in turn until one of them processes it. If the option is not handled **ber_sockbuf_ctrl**() will return 0.

Multiple **Sockbuf_IO** handlers can be stacked in multiple layers to provide various functionality. Currently defined layers include

LBER_SBIOD_LEVEL_PROVIDER
the lowest layer, talking directly to a network

LBER_SBIOD_LEVEL_TRANSPORT
an intermediate layer

LBER_SBIOD_LEVEL_APPLICATION
a higher layer

Currently defined **Sockbuf_IO** handlers in liblber include

ber_sockbuf_io_tcp
The default stream-oriented provider

ber_sockbuf_io_fd
A stream-oriented provider for local IPC sockets

ber_sockbuf_io_dgram
A datagram-oriented provider. This handler is only present if the liblber library was built with LDAP_CONNECTIONLESS defined.

ber_sockbuf_io_readahead
A buffering layer, usually used with a datagram provider to hide the datagram semantics from upper layers.

ber_sockbuf_io_debug
A generic handler that outputs hex dumps of all traffic. This handler may be inserted multiple times at arbitrary layers to show the flow of data between other handlers.

Additional handlers may be present in libldap if support for them was enabled:

ldap_pvt_sockbuf_io_sasl
An application layer handler for SASL encoding/decoding.

sb_tls_sbio
A transport layer handler for SSL/TLS encoding/decoding. Note that this handler is private to the library and is not exposed in the API.

The provided handlers are all instantiated implicitly by libldap, and applications generally will not need to directly manipulate them.

SEE ALSO

lber-decode(3), **lber-encode**(3), **lber-types**(3), **ldap_get_option**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ber_int_t, ber_uint_t, ber_len_t, ber_slen_t, ber_tag_t, struct berval, BerValue, BerVarray, BerElement, ber_bvfree, ber_bvecfree, ber_bvecadd, ber_bvarray_free, ber_bvarray_add, ber_bvdup, ber_dupbv, ber_bvstr, ber_bvstrdup, ber_str2bv, ber_alloc_t, ber_init, ber_init2, ber_free − OpenLDAP LBER types and allocation functions

LIBRARY

OpenLDAP LBER (liblber, -llber)

SYNOPSIS

#include <lber.h>

typedef impl_tag_t ber_tag_t;
typedef impl_int_t ber_int_t;
typedef impl_uint_t ber_uint_t;
typedef impl_len_t ber_len_t;
typedef impl_slen_t ber_slen_t;

typedef struct berval {
** ber_len_t bv_len;**
** char *bv_val;**
} BerValue, *BerVarray;

typedef struct berelement BerElement;

void ber_bvfree(struct berval * *bv***);**

void ber_bvecfree(struct berval ** *bvec***);**

void ber_bvecadd(struct berval *** *bvec*, **struct berval *** *bv***);**

void ber_bvarray_free(struct berval * *bvarray***);**

void ber_bvarray_add(BerVarray * *bvarray*, **BerValue *** *bv***);**

struct berval *ber_bvdup(const struct berval * *bv***);**

struct berval *ber_dupbv(const struct berval * *dst*, **struct berval *** *src***);**

struct berval *ber_bvstr(const char * *str***);**

struct berval *ber_bvstrdup(const char * *str***);**

struct berval *ber_str2bv(const char * *str*, **ber_len_t** *len*, **int** *dup*, **struct berval *** *bv***);**

BerElement *ber_alloc_t(int *options***);**

BerElement *ber_init(struct berval * *bv***);**

void ber_init2(BerElement * *ber*, **struct berval *** *bv*, **int** *options***);**

void ber_free(BerElement * *ber*, **int** *freebuf* **);**

DESCRIPTION

The following are the basic types and structures defined for use with the Lightweight BER library.

ber_int_t is a signed integer of at least 32 bits. It is commonly equivalent to **int**. **ber_uint_t** is the unsigned variant of **ber_int_t**.

ber_len_t is an unsigned integer of at least 32 bits used to represent a length. It is commonly equivalent to a **size_t**. **ber_slen_t** is the signed variant to **ber_len_t**.

ber_tag_t is an unsigned integer of at least 32 bits used to represent a BER tag. It is commonly equivalent to a **unsigned long**.

The actual definitions of the integral impl_TYPE_t types are platform specific.

BerValue, commonly used as **struct berval**, is used to hold an arbitrary sequence of octets. **bv_val** points

to **bv_len** octets. **bv_val** is not necessarily terminated by a NULL (zero) octet. **ber_bvfree**() frees a BerValue, pointed to by *bv*, returned from this API. If *bv* is NULL, the routine does nothing.

ber_bvecfree() frees an array of BerValues (and the array), pointed to by *bvec*, returned from this API. If *bvec* is NULL, the routine does nothing. **ber_bvecadd**() appends the *bv* pointer to the *bvec* array. Space for the array is allocated as needed. The end of the array is marked by a NULL pointer.

ber_bvarray_free() frees an array of BerValues (and the array), pointed to by *bvarray*, returned from this API. If *bvarray* is NULL, the routine does nothing. **ber_bvarray_add**() appends the contents of the BerValue pointed to by *bv* to the *bvarray* array. Space for the new element is allocated as needed. The end of the array is marked by a BerValue with a NULL bv_val field.

ber_bvdup() returns a copy of a BerValue. The routine returns NULL upon error (e.g. out of memory). The caller should use **ber_bvfree**() to deallocate the resulting BerValue. **ber_dupbv**() copies a BerValue from *src* to *dst*. If *dst* is NULL a new BerValue will be allocated to hold the copy. The routine returns NULL upon error, otherwise it returns a pointer to the copy. If *dst* is NULL the caller should use **ber_bvfree**() to deallocate the resulting BerValue, otherwise **ber_memfree**() should be used to deallocate the *dst->bv_val*. (The **ber_bvdup**() function is internally implemented as ber_dupbv(NULL, bv). **ber_bvdup**() is provided only for compatibility with an expired draft of the LDAP C API; **ber_dupbv**() is the preferred interface.)

ber_bvstr() returns a BerValue containing the string pointed to by *str*. **ber_bvstrdup**() returns a BerValue containing a copy of the string pointed to by *str*. **ber_str2bv**() returns a BerValue containing the string pointed to by *str*, whose length may be optionally specified in *len*. If *dup* is non-zero, the BerValue will contain a copy of *str*. If *len* is zero, the number of bytes to copy will be determined by **strlen**(3), otherwise *len* bytes will be copied. If *bv* is non-NULL, the result will be stored in the given BerValue, otherwise a new BerValue will be allocated to store the result. NOTE: Both **ber_bvstr**() and **ber_bvstrdup**() are implemented as macros using **ber_str2bv**() in this version of the library.

BerElement is an opaque structure used to maintain state information used in encoding and decoding. **ber_alloc_t**() is used to create an empty BerElement structure. If **LBER_USE_DER** is specified for the *options* parameter then data lengths for data written to the BerElement will be encoded in the minimal number of octets required, otherwise they will always be written as four byte values. **ber_init**() creates a BerElement structure that is initialized with a copy of the data in its *bv* parameter. **ber_init2**() initializes an existing BerElement *ber* using the data in the *bv* parameter. The data is referenced directly, not copied. The *options* parameter is the same as for **ber_alloc_t**(). **ber_free**() frees a BerElement pointed to by *ber*. If *ber* is NULL, the routine does nothing. If *freebuf* is zero, the internal buffer is not freed.

SEE ALSO

lber-encode(3), **lber-decode**(3), **lber-memory**(3)

ACKNOWLEDGEMENTS

NAME

ber_int_t, ber_uint_t, ber_len_t, ber_slen_t, ber_tag_t, struct berval, BerValue, BerVarray, BerElement, ber_bvfree, ber_bvecfree, ber_bvecadd, ber_bvarray_free, ber_bvarray_add, ber_bvdup, ber_dupbv, ber_bvstr, ber_bvstrdup, ber_str2bv, ber_alloc_t, ber_init, ber_init2, ber_free − OpenLDAP LBER types and allocation functions

LIBRARY

OpenLDAP LBER (liblber, -llber)

SYNOPSIS

#include <lber.h>

typedef impl_tag_t ber_tag_t;
typedef impl_int_t ber_int_t;
typedef impl_uint_t ber_uint_t;
typedef impl_len_t ber_len_t;
typedef impl_slen_t ber_slen_t;

typedef struct berval {
** ber_len_t bv_len;**
** char *bv_val;**
} BerValue, *BerVarray;

typedef struct berelement BerElement;

void ber_bvfree(struct berval * *bv***);**

void ber_bvecfree(struct berval ** *bvec***);**

void ber_bvecadd(struct berval *** *bvec*, **struct berval *** *bv***);**

void ber_bvarray_free(struct berval * *bvarray***);**

void ber_bvarray_add(BerVarray * *bvarray*, **BerValue *** *bv***);**

struct berval *ber_bvdup(const struct berval * *bv***);**

struct berval *ber_dupbv(const struct berval * *dst*, **struct berval *** *src***);**

struct berval *ber_bvstr(const char * *str***);**

struct berval *ber_bvstrdup(const char * *str***);**

struct berval *ber_str2bv(const char * *str*, **ber_len_t** *len*, **int** *dup*, **struct berval *** *bv***);**

BerElement *ber_alloc_t(int *options***);**

BerElement *ber_init(struct berval * *bv***);**

void ber_init2(BerElement * *ber*, **struct berval *** *bv*, **int** *options***);**

void ber_free(BerElement * *ber*, **int** *freebuf* **);**

DESCRIPTION

The following are the basic types and structures defined for use with the Lightweight BER library.

ber_int_t is a signed integer of at least 32 bits. It is commonly equivalent to **int**. **ber_uint_t** is the unsigned variant of **ber_int_t**.

ber_len_t is an unsigned integer of at least 32 bits used to represent a length. It is commonly equivalent to a **size_t**. **ber_slen_t** is the signed variant to **ber_len_t**.

ber_tag_t is an unsigned integer of at least 32 bits used to represent a BER tag. It is commonly equivalent to a **unsigned long**.

The actual definitions of the integral impl_TYPE_t types are platform specific.

BerValue, commonly used as **struct berval**, is used to hold an arbitrary sequence of octets. **bv_val** points

to **bv_len** octets. **bv_val** is not necessarily terminated by a NULL (zero) octet. **ber_bvfree**() frees a BerValue, pointed to by *bv*, returned from this API. If *bv* is NULL, the routine does nothing.

ber_bvecfree() frees an array of BerValues (and the array), pointed to by *bvec*, returned from this API. If *bvec* is NULL, the routine does nothing. **ber_bvecadd**() appends the *bv* pointer to the *bvec* array. Space for the array is allocated as needed. The end of the array is marked by a NULL pointer.

ber_bvarray_free() frees an array of BerValues (and the array), pointed to by *bvarray*, returned from this API. If *bvarray* is NULL, the routine does nothing. **ber_bvarray_add**() appends the contents of the BerValue pointed to by *bv* to the *bvarray* array. Space for the new element is allocated as needed. The end of the array is marked by a BerValue with a NULL bv_val field.

ber_bvdup() returns a copy of a BerValue. The routine returns NULL upon error (e.g. out of memory). The caller should use **ber_bvfree**() to deallocate the resulting BerValue. **ber_dupbv**() copies a BerValue from *src* to *dst*. If *dst* is NULL a new BerValue will be allocated to hold the copy. The routine returns NULL upon error, otherwise it returns a pointer to the copy. If *dst* is NULL the caller should use **ber_bvfree**() to deallocate the resulting BerValue, otherwise **ber_memfree**() should be used to deallocate the *dst->bv_val*. (The **ber_bvdup**() function is internally implemented as ber_dupbv(NULL, bv). **ber_bvdup**() is provided only for compatibility with an expired draft of the LDAP C API; **ber_dupbv**() is the preferred interface.)

ber_bvstr() returns a BerValue containing the string pointed to by *str*. **ber_bvstrdup**() returns a BerValue containing a copy of the string pointed to by *str*. **ber_str2bv**() returns a BerValue containing the string pointed to by *str*, whose length may be optionally specified in *len*. If *dup* is non-zero, the BerValue will contain a copy of *str*. If *len* is zero, the number of bytes to copy will be determined by **strlen**(3), otherwise *len* bytes will be copied. If *bv* is non-NULL, the result will be stored in the given BerValue, otherwise a new BerValue will be allocated to store the result. NOTE: Both **ber_bvstr**() and **ber_bvstrdup**() are implemented as macros using **ber_str2bv**() in this version of the library.

BerElement is an opaque structure used to maintain state information used in encoding and decoding. **ber_alloc_t**() is used to create an empty BerElement structure. If **LBER_USE_DER** is specified for the *options* parameter then data lengths for data written to the BerElement will be encoded in the minimal number of octets required, otherwise they will always be written as four byte values. **ber_init**() creates a BerElement structure that is initialized with a copy of the data in its *bv* parameter. **ber_init2**() initializes an existing BerElement *ber* using the data in the *bv* parameter. The data is referenced directly, not copied. The *options* parameter is the same as for **ber_alloc_t**(). **ber_free**() frees a BerElement pointed to by *ber*. If *ber* is NULL, the routine does nothing. If *freebuf* is zero, the internal buffer is not freed.

SEE ALSO

lber-encode(3), **lber-decode**(3), **lber-memory**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap − OpenLDAP Lightweight Directory Access Protocol API

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

DESCRIPTION

The Lightweight Directory Access Protocol (LDAP) (RFC 4510) provides access to X.500 directory services. These services may be stand−alone or part of a distributed directory service. This client API supports LDAP over TCP (RFC 4511), LDAP over TLS/SSL, and LDAP over IPC (UNIX domain sockets). This API supports SASL (RFC 4513) and Start TLS (RFC 4513) as well as a number of protocol extensions. This API is loosely based upon IETF/LDAPEXT C LDAP API draft specification, a (orphaned) work in progress.

The OpenLDAP Software package includes a stand−alone server in **slapd**(8), various LDAP clients, and an LDAP client library used to provide programmatic access to the LDAP protocol. This man page gives an overview of the LDAP library routines.

Both synchronous and asynchronous APIs are provided. Also included are various routines to parse the results returned from these routines. These routines are found in the −lldap library.

The basic interaction is as follows. A session handle is created using **ldap_initialize**(3) and set the protocol version to 3 by calling **ldap_set_option**(3). The underlying session is established first operation is issued. This would generally be a Start TLS or Bind operation, or a Search operation to read attributes of the Root DSE. A Start TLS operation is performed by calling **ldap_start_tls_s**(3). A LDAP bind operation is performed by calling **ldap_sasl_bind**(3) or one of its friends. A Search operation is performed by calling ldap_search_ext_s(3) or one of its friends.

Subsequently, additional operations are performed by calling one of the synchronous or asynchronous routines (e.g., **ldap_compare_ext_s**(3) or **ldap_compare_ext**(3) followed by **ldap_result**(3)). Results returned from these routines are interpreted by calling the LDAP parsing routines such as **ldap_parse_result**(3). The LDAP association and underlying connection is terminated by calling **ldap_unbind_ext**(3). Errors can be interpreted by calling **ldap_err2string**(3).

LDAP versions

This library supports version 3 of the Lightweight Directory Access Protocol (LDAPv3) as defined in RFC 4510. It also supports a variant of version 2 of LDAP as defined by U-Mich LDAP and, to some degree, RFC 1777. Version 2 (all variants) are considered obsolete. Version 3 should be used instead.

For backwards compatibility reasons, the library defaults to version 2. Hence, all new applications (and all actively maintained applications) should use **ldap_set_option**(3) to select version 3. The library manual pages assume version 3 has been selected.

INPUT and OUTPUT PARAMETERS

All character string input/output is expected to be/is UTF−8 encoded Unicode (version 3.2).

Distinguished names (DN) (and relative distinguished names (RDN) to be passed to the LDAP routines should conform to RFC 4514 UTF−8 string representation.

Search filters to be passed to the search routines are to be constructed by hand and should conform to RFC 4515 UTF−8 string representation.

LDAP URLs to be passed to routines are expected to conform to RFC 4516 format. The **ldap_url**(3) routines can be used to work with LDAP URLs.

LDAP controls to be passed to routines can be manipulated using the **ldap_controls**(3) routines.

DISPLAYING RESULTS

Results obtained from the search routines can be output by hand, by calling **ldap_first_entry**(3) and **ldap_next_entry**(3) to step through the entries returned, **ldap_first_attribute**(3) and

ldap_next_attribute(3) to step through an entry's attributes, and **ldap_get_values**(3) to retrieve a given attribute's values. Attribute values may or may not be displayable.

UTILITY ROUTINES

Also provided are various utility routines. The **ldap_sort**(3) routines are used to sort the entries and values returned via the ldap search routines.

DEPRECATED INTERFACES

A number of interfaces are now considered deprecated. For instance, ldap_add(3) is deprecated in favor of ldap_add_ext(3).

BER LIBRARY

Also included in the distribution is a set of lightweight Basic Encoding Rules routines. These routines are used by the LDAP library routines to encode and decode LDAP protocol elements using the (slightly simplified) Basic Encoding Rules defined by LDAP. They are not normally used directly by an LDAP application program except in the handling of controls and extended operations. The routines provide a printf and scanf−like interface, as well as lower−level access. These routines are discussed in **lber−decode**(3), **lber−encode**(3), **lber−memory**(3), and **lber−types**(3).

INDEX

ldap_initialize(3)	initialize the LDAP library without opening a connection to a server
ldap_result(3)	wait for the result from an asynchronous operation
ldap_abandon_ext(3)	abandon (abort) an asynchronous operation
ldap_add_ext(3)	asynchronously add an entry
ldap_add_ext_s(3)	synchronously add an entry
ldap_sasl_bind(3)	asynchronously bind to the directory
ldap_sasl_bind_s(3)	synchronously bind to the directory
ldap_unbind_ext(3)	synchronously unbind from the LDAP server and close the connection
ldap_unbind(3) and ldap_unbind_s(3) are	equivalent to **ldap_unbind_ext**(3)
ldap_memfree(3)	dispose of memory allocated by LDAP routines.
ldap_compare_ext(3)	asynchronously compare to a directory entry
ldap_compare_ext_s(3)	synchronously compare to a directory entry
ldap_delete_ext(3)	asynchronously delete an entry
ldap_delete_ext_s(3)	synchronously delete an entry
ld_errno(3)	LDAP error indication
ldap_errlist(3)	list of LDAP errors and their meanings
ldap_err2string(3)	convert LDAP error indication to a string
ldap_extended_operation(3)	asynchronously perform an arbitrary extended operation
ldap_extended_operation_s(3)	synchronously perform an arbitrary extended operation
ldap_first_attribute(3)	return first attribute name in an entry
ldap_next_attribute(3)	return next attribute name in an entry
ldap_first_entry(3)	return first entry in a chain of search results
ldap_next_entry(3)	return next entry in a chain of search results
ldap_count_entries(3)	return number of entries in a search result

ldap_get_dn(3)	extract the DN from an entry
ldap_get_values_len(3)	return an attribute's values with lengths
ldap_value_free_len(3)	free memory allocated by ldap_get_values_len(3)
ldap_count_values_len(3)	return number of values
ldap_modify_ext(3)	asynchronously modify an entry
ldap_modify_ext_s(3)	synchronously modify an entry
ldap_mods_free(3)	free array of pointers to mod structures used by ldap_modify_ext(3)
ldap_rename(3)	asynchronously rename an entry
ldap_rename_s(3)	synchronously rename an entry
ldap_msgfree(3)	free results allocated by ldap_result(3)
ldap_msgtype(3)	return the message type of a message from ldap_result(3)
ldap_msgid(3)	return the message id of a message from ldap_result(3)
ldap_search_ext(3)	asynchronously search the directory
ldap_search_ext_s(3)	synchronously search the directory
ldap_is_ldap_url(3)	check a URL string to see if it is an LDAP URL
ldap_url_parse(3)	break up an LDAP URL string into its components
ldap_sort_entries(3)	sort a list of search results
ldap_sort_values(3)	sort a list of attribute values
ldap_sort_strcasecmp(3)	case insensitive string comparison

SEE ALSO

ldap.conf(5), **slapd**(8), **draft-ietf-ldapext-ldap-c-api-xx.txt** <http://www.ietf.org>

ACKNOWLEDGEMENTS

These API manual pages are loosely based upon descriptions provided in the IETF/LDAPEXT C LDAP API Internet Draft, a (orphaned) work in progress.

NAME

ldap − OpenLDAP Lightweight Directory Access Protocol API

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

DESCRIPTION

The Lightweight Directory Access Protocol (LDAP) (RFC 4510) provides access to X.500 directory services. These services may be stand−alone or part of a distributed directory service. This client API supports LDAP over TCP (RFC 4511), LDAP over TLS/SSL, and LDAP over IPC (UNIX domain sockets). This API supports SASL (RFC 4513) and Start TLS (RFC 4513) as well as a number of protocol extensions. This API is loosely based upon IETF/LDAPEXT C LDAP API draft specification, a (orphaned) work in progress.

The OpenLDAP Software package includes a stand−alone server in **slapd**(8), various LDAP clients, and an LDAP client library used to provide programmatic access to the LDAP protocol. This man page gives an overview of the LDAP library routines.

Both synchronous and asynchronous APIs are provided. Also included are various routines to parse the results returned from these routines. These routines are found in the −lldap library.

The basic interaction is as follows. A session handle is created using **ldap_initialize**(3) and set the protocol version to 3 by calling **ldap_set_option**(3). The underlying session is established first operation is issued. This would generally be a Start TLS or Bind operation, or a Search operation to read attributes of the Root DSE. A Start TLS operation is performed by calling **ldap_start_tls_s**(3). A LDAP bind operation is performed by calling **ldap_sasl_bind**(3) or one of its friends. A Search operation is performed by calling ldap_search_ext_s(3) or one of its friends.

Subsequently, additional operations are performed by calling one of the synchronous or asynchronous routines (e.g., **ldap_compare_ext_s**(3) or **ldap_compare_ext**(3) followed by **ldap_result**(3)). Results returned from these routines are interpreted by calling the LDAP parsing routines such as **ldap_parse_result**(3). The LDAP association and underlying connection is terminated by calling **ldap_unbind_ext**(3). Errors can be interpreted by calling **ldap_err2string**(3).

LDAP versions

This library supports version 3 of the Lightweight Directory Access Protocol (LDAPv3) as defined in RFC 4510. It also supports a variant of version 2 of LDAP as defined by U-Mich LDAP and, to some degree, RFC 1777. Version 2 (all variants) are considered obsolete. Version 3 should be used instead.

For backwards compatibility reasons, the library defaults to version 2. Hence, all new applications (and all actively maintained applications) should use **ldap_set_option**(3) to select version 3. The library manual pages assume version 3 has been selected.

INPUT and OUTPUT PARAMETERS

All character string input/output is expected to be/is UTF−8 encoded Unicode (version 3.2).

Distinguished names (DN) (and relative distinguished names (RDN) to be passed to the LDAP routines should conform to RFC 4514 UTF−8 string representation.

Search filters to be passed to the search routines are to be constructed by hand and should conform to RFC 4515 UTF−8 string representation.

LDAP URLs to be passed to routines are expected to conform to RFC 4516 format. The **ldap_url**(3) routines can be used to work with LDAP URLs.

LDAP controls to be passed to routines can be manipulated using the **ldap_controls**(3) routines.

DISPLAYING RESULTS

Results obtained from the search routines can be output by hand, by calling **ldap_first_entry**(3) and **ldap_next_entry**(3) to step through the entries returned, **ldap_first_attribute**(3) and

ldap_next_attribute(3) to step through an entry's attributes, and **ldap_get_values**(3) to retrieve a given attribute's values. Attribute values may or may not be displayable.

UTILITY ROUTINES

Also provided are various utility routines. The **ldap_sort**(3) routines are used to sort the entries and values returned via the ldap search routines.

DEPRECATED INTERFACES

A number of interfaces are now considered deprecated. For instance, ldap_add(3) is deprecated in favor of ldap_add_ext(3). Deprecated interfaces generally remain in the library. The macro LDAP_DEPRECATED can be defined to a non-zero value (e.g., -DLDAP_DEPRECATED=1) when compiling program designed to use deprecated interfaces. It is recommended that developers writing new programs, or updating old programs, avoid use of deprecated interfaces. Over time, it is expected that documentation (and, eventually, support) for deprecated interfaces to be eliminated.

BER LIBRARY

Also included in the distribution is a set of lightweight Basic Encoding Rules routines. These routines are used by the LDAP library routines to encode and decode LDAP protocol elements using the (slightly simplified) Basic Encoding Rules defined by LDAP. They are not normally used directly by an LDAP application program except in the handling of controls and extended operations. The routines provide a printf and scanf−like interface, as well as lower−level access. These routines are discussed in **lber−decode**(3), **lber−encode**(3), **lber−memory**(3), and **lber−types**(3).

INDEX

ldap_initialize(3)	initialize the LDAP library without opening a connection to a server
ldap_result(3)	wait for the result from an asynchronous operation
ldap_abandon_ext(3)	abandon (abort) an asynchronous operation
ldap_add_ext(3)	asynchronously add an entry
ldap_add_ext_s(3)	synchronously add an entry
ldap_sasl_bind(3)	asynchronously bind to the directory
ldap_sasl_bind_s(3)	synchronously bind to the directory
ldap_unbind_ext(3)	synchronously unbind from the LDAP server and close the connection
ldap_unbind(3) and ldap_unbind_s(3) are	equivalent to **ldap_unbind_ext**(3)
ldap_memfree(3)	dispose of memory allocated by LDAP routines.
ldap_compare_ext(3)	asynchronously compare to a directory entry
ldap_compare_ext_s(3)	synchronously compare to a directory entry
ldap_delete_ext(3)	asynchronously delete an entry
ldap_delete_ext_s(3)	synchronously delete an entry
ld_errno(3)	LDAP error indication
ldap_errlist(3)	list of LDAP errors and their meanings
ldap_err2string(3)	convert LDAP error indication to a string
ldap_extended_operation(3)	asynchronously perform an arbitrary extended operation
ldap_extended_operation_s(3)	synchronously perform an arbitrary extended operation
ldap_first_attribute(3)	return first attribute name in an entry
ldap_next_attribute(3)	return next attribute name in an entry

ldap_first_entry(3)	return first entry in a chain of search results
ldap_next_entry(3)	return next entry in a chain of search results
ldap_count_entries(3)	return number of entries in a search result
ldap_get_dn(3)	extract the DN from an entry
ldap_get_values_len(3)	return an attribute's values with lengths
ldap_value_free_len(3)	free memory allocated by ldap_get_values_len(3)
ldap_count_values_len(3)	return number of values
ldap_modify_ext(3)	asynchronously modify an entry
ldap_modify_ext_s(3)	synchronously modify an entry
ldap_mods_free(3)	free array of pointers to mod structures used by ldap_modify_ext(3)
ldap_rename(3)	asynchronously rename an entry
ldap_rename_s(3)	synchronously rename an entry
ldap_msgfree(3)	free results allocated by ldap_result(3)
ldap_msgtype(3)	return the message type of a message from ldap_result(3)
ldap_msgid(3)	return the message id of a message from ldap_result(3)
ldap_search_ext(3)	asynchronously search the directory
ldap_search_ext_s(3)	synchronously search the directory
ldap_is_ldap_url(3)	check a URL string to see if it is an LDAP URL
ldap_url_parse(3)	break up an LDAP URL string into its components
ldap_sort_entries(3)	sort a list of search results
ldap_sort_values(3)	sort a list of attribute values
ldap_sort_strcasecmp(3)	case insensitive string comparison

SEE ALSO

ldap.conf(5), **slapd**(8), **draft-ietf-ldapext-ldap-c-api-xx.txt** <http://www.ietf.org>

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

These API manual pages are loosely based upon descriptions provided in the IETF/LDAPEXT C LDAP API Internet Draft, a (orphaned) work in progress.

NAME

ldap_abandon_ext − Abandon an LDAP operation in progress

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_abandon_ext(
LDAP ****ld*,**
Bint *msgid***,**
LDAPControl *****sctrls*,**
LDAPControl *****cctrls* **);

DESCRIPTION

The **ldap_abandon_ext()** routine is used to send a LDAP Abandon request for an operation in progress. The *msgid* passed should be the message id of an outstanding LDAP operation, such as returned by **ldap_search_ext**(3).

ldap_abandon_ext() checks to see if the result of the operation has already come in. If it has, it deletes it from the queue of pending messages. If not, it sends an LDAP abandon request to the LDAP server.

The caller can expect that the result of an abandoned operation will not be returned from a future call to **ldap_result**(3).

ldap_abandon_ext() allows server and client controls to be passed in via the *sctrls* and *cctrls* parameters, respectively.

ldap_abandon_ext() returns a code indicating success or, in the case of failure, the nature of the failure. See **ldap_error**(3) for details.

DEPRECATED INTERFACES

The **ldap_abandon()** routine is deprecated in favor of the **ldap_abandon_ext()** routine.

SEE ALSO

ldap(3), **ldap_error**(3), **ldap_result**(3), **ldap_search_ext**(3)

ACKNOWLEDGEMENTS

NAME

ldap_abandon_ext − Abandon an LDAP operation in progress

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_abandon_ext(
LDAP ****ld*,**
Bint *msgid***,**
LDAPControl *****sctrls*,**
LDAPControl *****cctrls* **);

DESCRIPTION

The **ldap_abandon_ext()** routine is used to send a LDAP Abandon request for an operation in progress. The *msgid* passed should be the message id of an outstanding LDAP operation, such as returned by **ldap_search_ext**(3).

ldap_abandon_ext() checks to see if the result of the operation has already come in. If it has, it deletes it from the queue of pending messages. If not, it sends an LDAP abandon request to the LDAP server.

The caller can expect that the result of an abandoned operation will not be returned from a future call to **ldap_result**(3).

ldap_abandon_ext() allows server and client controls to be passed in via the *sctrls* and *cctrls* parameters, respectively.

ldap_abandon_ext() returns a code indicating success or, in the case of failure, the nature of the failure. See **ldap_error**(3) for details.

DEPRECATED INTERFACES

The **ldap_abandon()** routine is deprecated in favor of the **ldap_abandon_ext()** routine.

Deprecated interfaces generally remain in the library. The macro LDAP_DEPRECATED can be defined to a non-zero value (e.g., -DLDAP_DEPRECATED=1) when compiling program designed to use deprecated interfaces. It is recommended that developers writing new programs, or updating old programs, avoid use of deprecated interfaces. Over time, it is expected that documentation (and, eventually, support) for deprecated interfaces to be eliminated.

SEE ALSO

ldap(3), **ldap_error**(3), **ldap_result**(3), **ldap_search_ext**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_add_ext, ldap_add_ext_s − Perform an LDAP add operation

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_add_ext(
**LDAP ****ld,*
*const char *dn***,**
LDAPMod *****attrs*,**
LDAPControl *****sctrls*,**
LDAPControl *****cctrls*,**
int ****msgidp* **);

int ldap_add_ext_s(
LDAP ****ld*,**
const char ****dn*,**
LDAPMod *****attrs*,**
LDAPControl ****sctrls*,**
LDAPControl ****cctrls* **);

DESCRIPTION

The **ldap_add_ext_s()** routine is used to perform an LDAP add operation. It takes *dn*, the DN of the entry to add, and *attrs*, a null-terminated array of the entry's attributes. The LDAPMod structure is used to represent attributes, with the *mod_type* and *mod_values* fields being used as described under **ldap_modify_ext**(3), and the *ldap_op* field being used only if you need to specify the LDAP_MOD_BVALUES option. Otherwise, it should be set to zero.

Note that all entries except that specified by the last component in the given DN must already exist. **ldap_add_ext_s()** returns an code indicating success or, in the case of failure, indicating the nature of failure of the operation. See **ldap_error**(3) for more details.

The **ldap_add_ext()** routine works just like **ldap_add_ext_s()**, but it is asynchronous. It returns the message id of the request it initiated. The result of this operation can be obtained by calling **ldap_result**(3).

DEPRECATED INTERFACES

The **ldap_add**() and **ldap_add_s**() routines are deprecated in favor of the **ldap_add_ext**() and **ldap_add_ext_s**() routines, respectively.

SEE ALSO

ldap(3), **ldap_error**(3), **ldap_modify**(3)

ACKNOWLEDGEMENTS

NAME

ldap_add_ext, ldap_add_ext_s − Perform an LDAP add operation

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_add_ext(
**LDAP ****ld,*
*const char *dn***,**
LDAPMod *****attrs*,**
LDAPControl *****sctrls*,**
LDAPControl *****cctrls*,**
int ****msgidp* **);

int ldap_add_ext_s(
LDAP ****ld*,**
const char ****dn*,**
LDAPMod *****attrs*,**
LDAPControl ****sctrls*,**
LDAPControl ****cctrls* **);

DESCRIPTION

The **ldap_add_ext_s()** routine is used to perform an LDAP add operation. It takes *dn*, the DN of the entry to add, and *attrs*, a null-terminated array of the entry's attributes. The LDAPMod structure is used to represent attributes, with the *mod_type* and *mod_value*s fields being used as described under **ldap_modify_ext**(3), and the *ldap_op* field being used only if you need to specify the LDAP_MOD_BVALUES option. Otherwise, it should be set to zero.

Note that all entries except that specified by the last component in the given DN must already exist. **ldap_add_ext_s()** returns an code indicating success or, in the case of failure, indicating the nature of failure of the operation. See **ldap_error**(3) for more details.

The **ldap_add_ext()** routine works just like **ldap_add_ext_s()**, but it is asynchronous. It returns the message id of the request it initiated. The result of this operation can be obtained by calling **ldap_result**(3).

DEPRECATED INTERFACES

The **ldap_add**() and **ldap_add_s**() routines are deprecated in favor of the **ldap_add_ext**() and **ldap_add_ext_s**() routines, respectively.

Deprecated interfaces generally remain in the library. The macro LDAP_DEPRECATED can be defined to a non-zero value (e.g., -DLDAP_DEPRECATED=1) when compiling program designed to use deprecated interfaces. It is recommended that developers writing new programs, or updating old programs, avoid use of deprecated interfaces. Over time, it is expected that documentation (and, eventually, support) for deprecated interfaces to be eliminated.

SEE ALSO

ldap(3), **ldap_error**(3), **ldap_modify**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_bind, ldap_bind_s, ldap_simple_bind, ldap_simple_bind_s, ldap_sasl_bind, ldap_sasl_bind_s, ldap_sasl_interactive_bind_s, ldap_parse_sasl_bind_result, ldap_unbind, ldap_unbind_s, ldap_unbind_ext, ldap_unbind_ext_s, ldap_set_rebind_proc − LDAP bind routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_bind(LDAP ****ld*, const char *****who***, const char *****cred***,**
int *method***);**

int ldap_bind_s(LDAP ****ld*, const char *****who***, const char *****cred***,**
int *method***);**

int ldap_simple_bind(LDAP ****ld*, const char *****who***, const char *****passwd***);**

int ldap_simple_bind_s(LDAP ****ld*, const char *****who***, const char *****passwd***);**

int ldap_sasl_bind(LDAP ****ld*, const char *****dn***, const char *****mechanism***,**
struct berval *****cred*, LDAPControl *****sctrls***[],**
LDAPControl *****cctrls*[], int *****msgidp***);**

int ldap_sasl_bind_s(LDAP ****ld*, const char *****dn***, const char *****mechanism***,**
struct berval *****cred*, LDAPControl *****sctrls***[],**
LDAPControl *****cctrls*[], struct berval ****servercredp***);**

int ldap_parse_sasl_bind_result(LDAP ****ld*, LDAPMessage *****res***,**
struct berval ****servercredp*, int** *freeit***);**

int ldap_sasl_interactive_bind_s(LDAP ****ld*, const char *****dn***,**
const char *****mechs*,
LDAPControl *****sctrls*[], LDAPControl *****cctrls***[],
unsigned *flags***, LDAP_SASL_INTERACT_PROC *****interact***,
void *****defaults*);**

int (LDAP_SASL_INTERACT_PROC)(LDAP ****ld*, unsigned** *flags***, void *****defaults***, void *****sasl_interact***);**

int ldap_unbind(LDAP ****ld*);**

int ldap_unbind_s(LDAP ****ld*);**

int ldap_unbind_ext(LDAP ****ld*, LDAPControl *****sctrls***[],**
LDAPControl *****cctrls*[]);**

int ldap_unbind_ext_s(LDAP ****ld*, LDAPControl *****sctrls***[],**
LDAPControl *****cctrls*[]);**

int ldap_set_rebind_proc (LDAP ****ld*, LDAP_REBIND_PROC *****ldap_proc***, void *****params***);**

int (LDAP_REBIND_PROC)(LDAP ****ld*, LDAP_CONST char *****url***, ber_tag_t** *request***, ber_int_t** *msgid***, void *****params***);**

DESCRIPTION

These routines provide various interfaces to the LDAP bind operation. After an association with an LDAP server is made using **ldap_init**(3), an LDAP bind operation should be performed before other operations are attempted over the connection. An LDAP bind is required when using Version 2 of the LDAP protocol; it is optional for Version 3 but is usually needed due to security considerations.

There are three types of bind calls, ones providing simple authentication, ones providing SASL authentication, and general routines capable of doing either simple or SASL authentication.

SASL (Simple Authentication and Security Layer) that can negotiate one of many different kinds of authentication. Both synchronous and asynchronous versions of each variant of the bind call are provided. All routines take *ld* as their first parameter, as returned from **ldap_init**(3).

SIMPLE AUTHENTICATION

The simplest form of the bind call is **ldap_simple_bind_s()**. It takes the DN to bind as in *who*, and the userPassword associated with the entry in *passwd*. It returns an LDAP error indication (see **ldap_error**(3)). The **ldap_simple_bind()** call is asynchronous, taking the same parameters but only initiating the bind operation and returning the message id of the request it sent. The result of the operation can be obtained by a subsequent call to **ldap_result**(3).

GENERAL AUTHENTICATION

The **ldap_bind()** and **ldap_bind_s()** routines can be used when the authentication method to use needs to be selected at runtime. They both take an extra *method* parameter selecting the authentication method to use. It should be set to LDAP_AUTH_SIMPLE to select simple authentication. **ldap_bind()** returns the message id of the request it initiates. **ldap_bind_s()** returns an LDAP error indication.

SASL AUTHENTICATION

For SASL binds the server always ignores any provided DN, so the *dn* parameter should always be NULL. **ldap_sasl_bind_s**() sends a single SASL bind request with the given SASL *mechanism* and credentials in the *cred* parameter. The format of the credentials depends on the particular SASL mechanism in use. For mechanisms that provide mutual authentication the server's credentials will be returned in the *servercredp* parameter. The routine returns an LDAP error indication (see **ldap_error**(3)). The **ldap_sasl_bind**() call is asynchronous, taking the same parameters but only sending the request and returning the message id of the request it sent. The result of the operation can be obtained by a subsequent call to **ldap_result**(3). The result must be additionally parsed by **ldap_parse_sasl_bind_result**() to obtain any server credentials sent from the server.

Many SASL mechanisms require multiple message exchanges to perform a complete authentication. Applications should generally use **ldap_sasl_interactive_bind_s**() rather than calling the basic **ldap_sasl_bind**() functions directly. The *mechs* parameter should contain a space-separated list of candidate mechanisms to use. If this parameter is NULL or empty the library will query the supportedSASLMechanisms attribute from the server's rootDSE for the list of SASL mechanisms the server supports. The *flags* parameter controls the interaction used to retrieve any necessary SASL authentication parameters and should be one of:

LDAP_SASL_AUTOMATIC
: use defaults if available, prompt otherwise

LDAP_SASL_INTERACTIVE
: always prompt

LDAP_SASL_QUIET
: never prompt

The *interact* function uses the provided *defaults* to handle requests from the SASL library for particular authentication parameters. There is no defined format for the *defaults* information; it is up to the caller to use whatever format is appropriate for the supplied *interact* function. The *sasl_interact* parameter comes from the underlying SASL library. When used with Cyrus SASL this is an array of **sasl_interact_t** structures. The Cyrus SASL library will prompt for a variety of inputs, including:

SASL_CB_GETREALM
: the realm for the authentication attempt

SASL_CB_AUTHNAME
: the username to authenticate

SASL_CB_PASS
: the password for the provided username

SASL_CB_USER
: the username to use for proxy authorization

SASL_CB_NOECHOPROMPT
: generic prompt for input with input echoing disabled

SASL_CB_ECHOPROMPT
generic prompt for input with input echoing enabled

SASL_CB_LIST_END
indicates the end of the array of prompts

See the Cyrus SASL documentation for more details.

REBINDING

The **ldap_set_rebind_proc** function() sets the process to use for binding when an operation returns a referral. This function is used when an application needs to bind to another server in order to follow a referral or search continuation reference.

The function takes *ld*, the *rebind* function, and the *params*, the arbitrary data like state information which the client might need to properly rebind. The LDAP_OPT_REFERRALS option in the *ld* must be set to ON for the libraries to use the rebind function. Use the **ldap_set_option** function to set the value.

The rebind function parameters are as follows:

The *ld* parameter must be used by the application when binding to the referred server if the application wants the libraries to follow the referral.

The *url* parameter points to the URL referral string received from the LDAP server. The LDAP application can use the **ldap_url_parse**(3) function to parse the string into its components.

The *request* parameter specifies the type of request that generated the referral.

The *msgid* parameter specifies the message ID of the request generating the referral.

The *params* parameter is the same value as passed originally to the **ldap_set_rebind_proc**() function.

The LDAP libraries set all the parameters when they call the rebind function. The application should not attempt to free either the ld or the url structures in the rebind function.

The application must supply to the rebind function the required authentication information such as, user name, password, and certificates. The rebind function must use a synchronous bind method.

UNBINDING

The **ldap_unbind()** call is used to unbind from the directory, terminate the current association, and free the resources contained in the *ld* structure. Once it is called, the connection to the LDAP server is closed, and the *ld* structure is invalid. The **ldap_unbind_s()** call is just another name for **ldap_unbind()**; both of these calls are synchronous in nature.

The **ldap_unbind_ext()** and **ldap_unbind_ext_s()** allows the operations to specify controls.

ERRORS

Asynchronous routines will return -1 in case of error, setting the *ld_errno* parameter of the *ld* structure. Synchronous routines return whatever *ld_errno* is set to. See **ldap_error**(3) for more information.

NOTES

If an anonymous bind is sufficient for the application, the rebind process need not be provided. The LDAP libraries with the LDAP_OPT_REFERRALS option set to ON (default value) will automatically follow referrals using an anonymous bind.

If the application needs stronger authentication than an anonymous bind, you need to provide a rebind process for that authentication method. The bind method must be synchronous.

SEE ALSO

ldap(3), **ldap_error**(3), **ldap_open**(3), **ldap_set_option**(3), **ldap_url_parse**(3) **RFC 4422** (http://www.rfc-editor.org), **Cyrus SASL** (http://asg.web.cmu.edu/sasl/)

ACKNOWLEDGEMENTS

NAME

ldap_bind, ldap_bind_s, ldap_simple_bind, ldap_simple_bind_s, ldap_sasl_bind, ldap_sasl_bind_s, ldap_sasl_interactive_bind_s, ldap_parse_sasl_bind_result, ldap_unbind, ldap_unbind_s, ldap_unbind_ext, ldap_unbind_ext_s, ldap_set_rebind_proc − LDAP bind routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_bind(LDAP ****ld*, const char *****who***, const char *****cred***,**
int *method***);**

int ldap_bind_s(LDAP ****ld*, const char *****who***, const char *****cred***,**
int *method***);**

int ldap_simple_bind(LDAP ****ld*, const char *****who***, const char *****passwd***);**

int ldap_simple_bind_s(LDAP ****ld*, const char *****who***, const char *****passwd***);**

int ldap_sasl_bind(LDAP ****ld*, const char *****dn***, const char *****mechanism***,**
struct berval *****cred*, LDAPControl *****sctrls***[],**
LDAPControl *****cctrls*[], int *****msgidp***);**

int ldap_sasl_bind_s(LDAP ****ld*, const char *****dn***, const char *****mechanism***,**
struct berval *****cred*, LDAPControl *****sctrls***[],**
LDAPControl *****cctrls*[], struct berval ****servercredp***);**

int ldap_parse_sasl_bind_result(LDAP ****ld*, LDAPMessage *****res***,**
struct berval ****servercredp*, int** *freeit***);**

int ldap_sasl_interactive_bind_s(LDAP ****ld*, const char *****dn***,**
const char *****mechs*,
LDAPControl *****sctrls*[], LDAPControl *****cctrls***[],
unsigned *flags*, **LDAP_SASL_INTERACT_PROC *****interact***,
void *****defaults*);**

int (LDAP_SASL_INTERACT_PROC)(LDAP ****ld*, unsigned** *flags*, **void *****defaults***, void *****sasl_interact***);**

int ldap_unbind(LDAP ****ld*);**

int ldap_unbind_s(LDAP ****ld*);**

int ldap_unbind_ext(LDAP ****ld*, LDAPControl *****sctrls***[],**
LDAPControl *****cctrls*[]);**

int ldap_unbind_ext_s(LDAP ****ld*, LDAPControl *****sctrls***[],**
LDAPControl *****cctrls*[]);**

int ldap_set_rebind_proc (LDAP ****ld*, LDAP_REBIND_PROC *****ldap_proc***, void *****params***);**

int (LDAP_REBIND_PROC)(LDAP ****ld*, LDAP_CONST char *****url***, ber_tag_t** *request*, **ber_int_t** *msgid*, **void *****params***);**

DESCRIPTION

These routines provide various interfaces to the LDAP bind operation. After an association with an LDAP server is made using **ldap_init**(3), an LDAP bind operation should be performed before other operations are attempted over the connection. An LDAP bind is required when using Version 2 of the LDAP protocol; it is optional for Version 3 but is usually needed due to security considerations.

There are three types of bind calls, ones providing simple authentication, ones providing SASL authentication, and general routines capable of doing either simple or SASL authentication.

SASL (Simple Authentication and Security Layer) that can negotiate one of many different kinds of authentication. Both synchronous and asynchronous versions of each variant of the bind call are provided. All routines take *ld* as their first parameter, as returned from **ldap_init**(3).

SIMPLE AUTHENTICATION

The simplest form of the bind call is **ldap_simple_bind_s()**. It takes the DN to bind as in *who*, and the userPassword associated with the entry in *passwd*. It returns an LDAP error indication (see **ldap_error**(3)). The **ldap_simple_bind()** call is asynchronous, taking the same parameters but only initiating the bind operation and returning the message id of the request it sent. The result of the operation can be obtained by a subsequent call to **ldap_result**(3).

GENERAL AUTHENTICATION

The **ldap_bind()** and **ldap_bind_s()** routines can be used when the authentication method to use needs to be selected at runtime. They both take an extra *method* parameter selecting the authentication method to use. It should be set to LDAP_AUTH_SIMPLE to select simple authentication. **ldap_bind()** returns the message id of the request it initiates. **ldap_bind_s()** returns an LDAP error indication.

SASL AUTHENTICATION

For SASL binds the server always ignores any provided DN, so the *dn* parameter should always be NULL. **ldap_sasl_bind_s**() sends a single SASL bind request with the given SASL *mechanism* and credentials in the *cred* parameter. The format of the credentials depends on the particular SASL mechanism in use. For mechanisms that provide mutual authentication the server's credentials will be returned in the *servercredp* parameter. The routine returns an LDAP error indication (see **ldap_error**(3)). The **ldap_sasl_bind**() call is asynchronous, taking the same parameters but only sending the request and returning the message id of the request it sent. The result of the operation can be obtained by a subsequent call to **ldap_result**(3). The result must be additionally parsed by **ldap_parse_sasl_bind_result**() to obtain any server credentials sent from the server.

Many SASL mechanisms require multiple message exchanges to perform a complete authentication. Applications should generally use **ldap_sasl_interactive_bind_s**() rather than calling the basic **ldap_sasl_bind**() functions directly. The *mechs* parameter should contain a space-separated list of candidate mechanisms to use. If this parameter is NULL or empty the library will query the supportedSASLMechanisms attribute from the server's rootDSE for the list of SASL mechanisms the server supports. The *flags* parameter controls the interaction used to retrieve any necessary SASL authentication parameters and should be one of:

LDAP_SASL_AUTOMATIC
: use defaults if available, prompt otherwise

LDAP_SASL_INTERACTIVE
: always prompt

LDAP_SASL_QUIET
: never prompt

The *interact* function uses the provided *defaults* to handle requests from the SASL library for particular authentication parameters. There is no defined format for the *defaults* information; it is up to the caller to use whatever format is appropriate for the supplied *interact* function. The *sasl_interact* parameter comes from the underlying SASL library. When used with Cyrus SASL this is an array of **sasl_interact_t** structures. The Cyrus SASL library will prompt for a variety of inputs, including:

SASL_CB_GETREALM
: the realm for the authentication attempt

SASL_CB_AUTHNAME
: the username to authenticate

SASL_CB_PASS
: the password for the provided username

SASL_CB_USER
: the username to use for proxy authorization

SASL_CB_NOECHOPROMPT
: generic prompt for input with input echoing disabled

SASL_CB_ECHOPROMPT
generic prompt for input with input echoing enabled

SASL_CB_LIST_END
indicates the end of the array of prompts

See the Cyrus SASL documentation for more details.

REBINDING

The **ldap_set_rebind_proc** function() sets the process to use for binding when an operation returns a referral. This function is used when an application needs to bind to another server in order to follow a referral or search continuation reference.

The function takes *ld*, the *rebind* function, and the *params*, the arbitrary data like state information which the client might need to properly rebind. The LDAP_OPT_REFERRALS option in the *ld* must be set to ON for the libraries to use the rebind function. Use the **ldap_set_option** function to set the value.

The rebind function parameters are as follows:

The *ld* parameter must be used by the application when binding to the referred server if the application wants the libraries to follow the referral.

The *url* parameter points to the URL referral string received from the LDAP server. The LDAP application can use the **ldap_url_parse**(3) function to parse the string into its components.

The *request* parameter specifies the type of request that generated the referral.

The *msgid* parameter specifies the message ID of the request generating the referral.

The *params* parameter is the same value as passed originally to the **ldap_set_rebind_proc**() function.

The LDAP libraries set all the parameters when they call the rebind function. The application should not attempt to free either the ld or the url structures in the rebind function.

The application must supply to the rebind function the required authentication information such as, user name, password, and certificates. The rebind function must use a synchronous bind method.

UNBINDING

The **ldap_unbind()** call is used to unbind from the directory, terminate the current association, and free the resources contained in the *ld* structure. Once it is called, the connection to the LDAP server is closed, and the *ld* structure is invalid. The **ldap_unbind_s()** call is just another name for **ldap_unbind()**; both of these calls are synchronous in nature.

The **ldap_unbind_ext()** and **ldap_unbind_ext_s()** allows the operations to specify controls.

ERRORS

Asynchronous routines will return -1 in case of error, setting the *ld_errno* parameter of the *ld* structure. Synchronous routines return whatever *ld_errno* is set to. See **ldap_error**(3) for more information.

NOTES

If an anonymous bind is sufficient for the application, the rebind process need not be provided. The LDAP libraries with the LDAP_OPT_REFERRALS option set to ON (default value) will automatically follow referrals using an anonymous bind.

If the application needs stronger authentication than an anonymous bind, you need to provide a rebind process for that authentication method. The bind method must be synchronous.

SEE ALSO

ldap(3), **ldap_error**(3), **ldap_open**(3), **ldap_set_option**(3), **ldap_url_parse**(3) **RFC 4422** (http://www.rfc-editor.org), **Cyrus SASL** (http://asg.web.cmu.edu/sasl/)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_compare, ldap_compare_s, ldap_compare_ext, ldap_compare_ext_s − Perform an LDAP compare operation.

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_compare_ext(
LDAP ****ld*,**
char ****dn*,**
char ****attr*,**
const struct berval ****bvalue*,**
LDAPControl *****serverctrls*,**
LDAPControl *****clientctrls*,**
int ****msgidp* **);

int ldap_compare_ext_s(
LDAP ****ld*,**
char ****dn*,**
char ****attr*,**
const struct berval ****bvalue*,**
LDAPControl *****serverctrls*,**
LDAPControl *****clientctrls* **);

DESCRIPTION

The **ldap_compare_ext_s()** routine is used to perform an LDAP compare operation synchronously. It takes *dn*, the DN of the entry upon which to perform the compare, and *attr* and *value*, the attribute description and value to compare to those found in the entry. It returns a code, which will be LDAP_COMPARE_TRUE if the entry contains the attribute value and LDAP_COMPARE_FALSE if it does not. Otherwise, an error code is returned that indicates the nature of the problem. See **ldap**(3) for details.

The **ldap_compare_ext()** routine is used to perform an LDAP compare operation asynchronously. It takes the same parameters as **ldap_compare_ext_s()**, but provides the message id of the request it initiated in the integer pointed to *msgidp*. The result of the compare can be obtained by a subsequent call to **ldap_result**(3).

Both routines allow server and client controls to be specified to extend the compare request.

DEPRECATED INTERFACES

The routines **ldap_compare**() and **ldap_compare_s**() are deprecated in favor of **ldap_compare_ext**() and **ldap_compare_ext_s**(), respectively.

SEE ALSO

ldap(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

NAME

ldap_compare, ldap_compare_s, ldap_compare_ext, ldap_compare_ext_s − Perform an LDAP compare operation.

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_compare_ext(
LDAP ****ld*,**
char ****dn*,**
char ****attr*,**
const struct berval ****bvalue*,**
LDAPControl *****serverctrls*,**
LDAPControl *****clientctrls*,**
int ****msgidp* **);

int ldap_compare_ext_s(
LDAP ****ld*,**
char ****dn*,**
char ****attr*,**
const struct berval ****bvalue*,**
LDAPControl *****serverctrls*,**
LDAPControl *****clientctrls* **);

DESCRIPTION

The **ldap_compare_ext_s()** routine is used to perform an LDAP compare operation synchronously. It takes *dn*, the DN of the entry upon which to perform the compare, and *attr* and *value*, the attribute description and value to compare to those found in the entry. It returns a code, which will be LDAP_COMPARE_TRUE if the entry contains the attribute value and LDAP_COMPARE_FALSE if it does not. Otherwise, an error code is returned that indicates the nature of the problem. See **ldap**(3) for details.

The **ldap_compare_ext()** routine is used to perform an LDAP compare operation asynchronously. It takes the same parameters as **ldap_compare_ext_s()**, but provides the message id of the request it initiated in the integer pointed to *msgidp*. The result of the compare can be obtained by a subsequent call to **ldap_result**(3).

Both routines allow server and client controls to be specified to extend the compare request.

DEPRECATED INTERFACES

The routines **ldap_compare**() and **ldap_compare_s**() are deprecated in favor of **ldap_compare_ext**() and **ldap_compare_ext_s**(), respectively.

Deprecated interfaces generally remain in the library. The macro LDAP_DEPRECATED can be defined to a non-zero value (e.g., -DLDAP_DEPRECATED=1) when compiling program designed to use deprecated interfaces. It is recommended that developers writing new programs, or updating old programs, avoid use of deprecated interfaces. Over time, it is expected that documentation (and, eventually, support) for deprecated interfaces to be eliminated.

SEE ALSO

ldap(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_control_create, ldap_control_find, ldap_control_dup, ldap_controls_dup, ldap_control_free, ldap_controls_free − LDAP control manipulation routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_control_create(const char ****oid*, int** *iscritical***, struct berval *****value***, int** *dupval***, LDAPControl ****ctrlp***);**

LDAPControl *ldap_control_find(const char *****oid*, LDAPControl ****ctrls***, LDAPControl *****nextctrlp***);**

LDAPControl *ldap_control_dup(LDAPControl *****ctrl*);**

LDAPControl **ldap_controls_dup(LDAPControl ****ctrls*);**

void ldap_control_free(LDAPControl *****ctrl*);**

void ldap_controls_free(LDAPControl ****ctrls*);**

DESCRIPTION

These routines are used to manipulate structures used for LDAP controls.

ldap_control_create() creates a control with the specified *OID* using the contents of the *value* parameter for the control value, if any. The content of *value* is duplicated if *dupval* is non-zero. The *iscritical* parameter must be non-zero for a critical control. The created control is returned in the *ctrlp* parameter. The routine returns **LDAP_SUCCESS** on success or some other error code on failure. The content of *value*, for supported control types, can be prepared using helpers provided by this implementation of libldap, usually in the form **ldap_create_<control name>_control_value**(). Otherwise, it can be BER-encoded using the functionalities of liblber.

ldap_control_find() searches the NULL-terminated *ctrls* array for a control whose OID matches the *oid* parameter. The routine returns a pointer to the control if found, NULL otherwise. If the parameter *nextctrlp* is not NULL, on return it will point to the next control in the array, and can be passed to the **ldap_control_find**() routine for subsequent calls, to find further occurrences of the same control type. The use of this function is discouraged; the recommended way of handling controls in responses consists in going through the array of controls, dealing with each of them in the returned order, since it could matter.

ldap_control_dup() duplicates an individual control structure, and **ldap_controls_dup**() duplicates a NULL-terminated array of controls.

ldap_control_free() frees an individual control structure, and **ldap_controls_free**() frees a NULL-terminated array of controls.

SEE ALSO

ldap(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

NAME

ldap_control_create, ldap_control_find, ldap_control_dup, ldap_controls_dup, ldap_control_free, ldap_controls_free − LDAP control manipulation routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_control_create(const char ****oid*, int** *iscritical***, struct berval *****value***, int** *dupval***, LDAPControl ****ctrlp***);**

LDAPControl *ldap_control_find(const char ****oid*, LDAPControl ****ctrls***, LDAPControl *****nextctrlp***);**

LDAPControl *ldap_control_dup(LDAPControl *****ctrl*);**

LDAPControl **ldap_controls_dup(LDAPControl ****ctrls*);**

void ldap_control_free(LDAPControl *****ctrl*);**

void ldap_controls_free(LDAPControl ****ctrls*);**

DESCRIPTION

These routines are used to manipulate structures used for LDAP controls.

ldap_control_create() creates a control with the specified *OID* using the contents of the *value* parameter for the control value, if any. The content of *value* is duplicated if *dupval* is non-zero. The *iscritical* parameter must be non-zero for a critical control. The created control is returned in the *ctrlp* parameter. The routine returns **LDAP_SUCCESS** on success or some other error code on failure. The content of *value*, for supported control types, can be prepared using helpers provided by this implementation of libldap, usually in the form **ldap_create_<control name>_control_value**(). Otherwise, it can be BER-encoded using the functionalities of liblber.

ldap_control_find() searches the NULL-terminated *ctrls* array for a control whose OID matches the *oid* parameter. The routine returns a pointer to the control if found, NULL otherwise. If the parameter *nextctrlp* is not NULL, on return it will point to the next control in the array, and can be passed to the **ldap_control_find**() routine for subsequent calls, to find further occurrences of the same control type. The use of this function is discouraged; the recommended way of handling controls in responses consists in going through the array of controls, dealing with each of them in the returned order, since it could matter.

ldap_control_dup() duplicates an individual control structure, and **ldap_controls_dup**() duplicates a NULL-terminated array of controls.

ldap_control_free() frees an individual control structure, and **ldap_controls_free**() frees a NULL-terminated array of controls.

SEE ALSO

ldap(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_delete, ldap_delete_s, ldap_delete_ext, ldap_delete_ext_s − Perform an LDAP delete operation.

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_delete_s(ld, dn)
LDAP *ld;
char *dn;

int ldap_delete(ld, dn)
LDAP *ld;
char *dn;

int ldap_delete_ext(ld, dn, serverctrls, clientctrls, msgidp)
LDAP *ld;
char *dn;
LDAPControl **serverctrls, **clientctrls;
int *msgidp;

int ldap_delete_ext_s(ld, dn, serverctrls, clientctrls)
LDAP *ld;
char *dn;
LDAPControl **serverctrls, **clientctrls;

DESCRIPTION

The **ldap_delete_s()** routine is used to perform an LDAP delete operation synchronously. It takes *dn*, the DN of the entry to be deleted. It returns an LDAP error code, indicating the success or failure of the operation.

The **ldap_delete()** routine is used to perform an LDAP delete operation asynchronously. It takes the same parameters as **ldap_delete_s(),** but returns the message id of the request it initiated. The result of the delete can be obtained by a subsequent call to **ldap_result**(3).

The **ldap_delete_ext()** routine allows server and client controls to be specified to extend the delete request. This routine is asynchronous like ldap_delete(), but its return value is an LDAP error code. It stores the message id of the request in the integer pointed to by msgidp.

The **ldap_delete_ext_s()** routine is the synchronous version of **ldap_delete_ext().** It also returns an LDAP error code indicating success or failure of the operation.

ERRORS

ldap_delete_s() returns an LDAP error code which can be interpreted by calling one of **ldap_perror**(3) and friends. **ldap_delete()** returns -1 if something went wrong initiating the request. It returns the non-negative message id of the request if things went ok.

ldap_delete_ext() and **ldap_delete_ext_s()** return some Non-zero value if something went wrong initiating the request, else return 0.

SEE ALSO

ldap(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

NAME

ldap_delete, ldap_delete_s, ldap_delete_ext, ldap_delete_ext_s − Perform an LDAP delete operation.

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_delete_s(ld, dn)
LDAP *ld;
char *dn;

int ldap_delete(ld, dn)
LDAP *ld;
char *dn;

int ldap_delete_ext(ld, dn, serverctrls, clientctrls, msgidp)
LDAP *ld;
char *dn;
LDAPControl **serverctrls, **clientctrls;
int *msgidp;

int ldap_delete_ext_s(ld, dn, serverctrls, clientctrls)
LDAP *ld;
char *dn;
LDAPControl **serverctrls, **clientctrls;

DESCRIPTION

The **ldap_delete_s()** routine is used to perform an LDAP delete operation synchronously. It takes *dn*, the DN of the entry to be deleted. It returns an LDAP error code, indicating the success or failure of the operation.

The **ldap_delete()** routine is used to perform an LDAP delete operation asynchronously. It takes the same parameters as **ldap_delete_s(),** but returns the message id of the request it initiated. The result of the delete can be obtained by a subsequent call to **ldap_result**(3).

The **ldap_delete_ext()** routine allows server and client controls to be specified to extend the delete request. This routine is asynchronous like ldap_delete(), but its return value is an LDAP error code. It stores the message id of the request in the integer pointed to by msgidp.

The **ldap_delete_ext_s()** routine is the synchronous version of **ldap_delete_ext().** It also returns an LDAP error code indicating success or failure of the operation.

ERRORS

ldap_delete_s() returns an LDAP error code which can be interpreted by calling one of **ldap_perror**(3) and friends. **ldap_delete()** returns -1 if something went wrong initiating the request. It returns the non-negative message id of the request if things went ok.

ldap_delete_ext() and **ldap_delete_ext_s()** return some Non-zero value if something went wrong initiating the request, else return 0.

SEE ALSO

ldap(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_perror, ld_errno, ldap_result2error, ldap_errlist, ldap_err2string − LDAP protocol error handling routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

char *ldap_err2string(int *err* **);**

DESCRIPTION

The **ldap_err2string()** routine provides short description of the various codes returned by routines in this library. The returned string is a pointer to a static area that should not be modified.

These codes are either negative, indicating an API error code; positive, indicating an LDAP resultCode other than ´success' (0), or - zero, indicating both successful use of the API and the LDAP resultCode ´success' (0).

The code associated with an LDAP session is accessible using **ldap_get_option**(3) and **ldap_set_option**(3) with the **LDAP_OPT_RESULT_CODE** option (previously called **LDAP_OPT_ERROR_NUMBER**).

PROTOCOL RESULT CODES

This section provides a partial list of protocol codes recognized by the library. As LDAP is extensible, additional values may be returned. A complete listing of *registered* LDAP result codes can be obtained from the *Internet Assigned Numbers Authority* <http://www.iana.org>.

LDAP_SUCCESS The request was successful.

LDAP_OPERATIONS_ERROR
An operations error occurred.

LDAP_PROTOCOL_ERROR
A protocol violation was detected.

LDAP_TIMELIMIT_EXCEEDED
An LDAP time limit was exceeded.

LDAP_SIZELIMIT_EXCEEDED
An LDAP size limit was exceeded.

LDAP_COMPARE_FALSE
A compare operation returned false.

LDAP_COMPARE_TRUE
A compare operation returned true.

LDAP_STRONG_AUTH_NOT_SUPPORTED
The LDAP server does not support strong authentication.

LDAP_STRONG_AUTH_REQUIRED
Strong authentication is required for the operation.

LDAP_PARTIAL_RESULTS
Partial results only returned.

LDAP_NO_SUCH_ATTRIBUTE
The attribute type specified does not exist in the entry.

LDAP_UNDEFINED_TYPE
: The attribute type specified is invalid.

LDAP_INAPPROPRIATE_MATCHING
: Filter type not supported for the specified attribute.

LDAP_CONSTRAINT_VIOLATION
: An attribute value specified violates some constraint (e.g., a postalAddress has too many lines, or a line that is too long).

LDAP_TYPE_OR_VALUE_EXISTS
: An attribute type or attribute value specified already exists in the entry.

LDAP_INVALID_SYNTAX
: An invalid attribute value was specified.

LDAP_NO_SUCH_OBJECT
: The specified object does not exist in The Directory.

LDAP_ALIAS_PROBLEM
: An alias in The Directory points to a nonexistent entry.

LDAP_INVALID_DN_SYNTAX
: A syntactically invalid DN was specified.

LDAP_IS_LEAF
: The object specified is a leaf.

LDAP_ALIAS_DEREF_PROBLEM
: A problem was encountered when dereferencing an alias.

LDAP_INAPPROPRIATE_AUTH
: Inappropriate authentication was specified (e.g., LDAP_AUTH_SIMPLE was specified and the entry does not have a userPassword attribute).

LDAP_INVALID_CREDENTIALS
: Invalid credentials were presented (e.g., the wrong password).

LDAP_INSUFFICIENT_ACCESS
: The user has insufficient access to perform the operation.

LDAP_BUSY
: The DSA is busy.

LDAP_UNAVAILABLE
: The DSA is unavailable.

LDAP_UNWILLING_TO_PERFORM
: The DSA is unwilling to perform the operation.

LDAP_LOOP_DETECT
: A loop was detected.

LDAP_NAMING_VIOLATION
: A naming violation occurred.

LDAP_OBJECT_CLASS_VIOLATION
: An object class violation occurred (e.g., a "must" attribute was missing from the entry).

LDAP_NOT_ALLOWED_ON_NONLEAF
: The operation is not allowed on a nonleaf object.

LDAP_NOT_ALLOWED_ON_RDN
: The operation is not allowed on an RDN.

LDAP_ALREADY_EXISTS
: The entry already exists.

LDAP_NO_OBJECT_CLASS_MODS
: Object class modifications are not allowed.

LDAP_OTHER An unknown error occurred.

API ERROR CODES

This section provides a complete list of API error codes recognized by the library. Note that LDAP_SUCCESS indicates success of an API call in addition to representing the return of the LDAP ´success' resultCode.

LDAP_SERVER_DOWN The LDAP library can't contact the LDAP server.

LDAP_LOCAL_ERROR Some local error occurred. This is usually a failed dynamic memory allocation.

LDAP_ENCODING_ERROR
An error was encountered encoding parameters to send to the LDAP server.

LDAP_DECODING_ERROR
An error was encountered decoding a result from the LDAP server.

LDAP_TIMEOUT A timelimit was exceeded while waiting for a result.

LDAP_AUTH_UNKNOWN
The authentication method specified to ldap_bind() is not known.

LDAP_FILTER_ERROR An invalid filter was supplied to ldap_search() (e.g., unbalanced parentheses).

LDAP_PARAM_ERROR An ldap routine was called with a bad parameter.

LDAP_NO_MEMORY An memory allocation (e.g., malloc(3) or other dynamic memory allocator) call failed in an ldap library routine.

LDAP_USER_CANCELED
Indicates the user cancelled the operation.

LDAP_CONNECT_ERROR
Indicates a connection problem.

LDAP_NOT_SUPPORTED
Indicates the routine was called in a manner not supported by the library.

LDAP_CONTROL_NOT_FOUND
Indicates the control provided is unknown to the client library.

LDAP_NO_RESULTS_RETURNED
Indicates no results returned.

LDAP_MORE_RESULTS_TO_RETURN
Indicates more results could be returned.

LDAP_CLIENT_LOOP Indicates the library has detected a loop in its processing.

LDAP_REFERRAL_LIMIT_EXCEEDED
Indicates the referral limit has been exceeded.

DEPRECATED

SEE ALSO

ldap(3),

ACKNOWLEDGEMENTS

NAME

ldap_perror, ld_errno, ldap_result2error, ldap_errlist, ldap_err2string − LDAP protocol error handling routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

char *ldap_err2string(int *err* **);**

DESCRIPTION

The **ldap_err2string()** routine provides short description of the various codes returned by routines in this library. The returned string is a pointer to a static area that should not be modified.

These codes are either negative, indicating an API error code; positive, indicating an LDAP resultCode other than ´success' (0), or - zero, indicating both successful use of the API and the LDAP resultCode ´success' (0).

The code associated with an LDAP session is accessible using **ldap_get_option**(3) and **ldap_set_option**(3) with the **LDAP_OPT_RESULT_CODE** option (previously called **LDAP_OPT_ERROR_NUMBER**).

PROTOCOL RESULT CODES

This section provides a partial list of protocol codes recognized by the library. As LDAP is extensible, additional values may be returned. A complete listing of *registered* LDAP result codes can be obtained from the *Internet Assigned Numbers Authority* <http://www.iana.org>.

LDAP_SUCCESS The request was successful.

LDAP_OPERATIONS_ERROR
An operations error occurred.

LDAP_PROTOCOL_ERROR
A protocol violation was detected.

LDAP_TIMELIMIT_EXCEEDED
An LDAP time limit was exceeded.

LDAP_SIZELIMIT_EXCEEDED
An LDAP size limit was exceeded.

LDAP_COMPARE_FALSE
A compare operation returned false.

LDAP_COMPARE_TRUE
A compare operation returned true.

LDAP_STRONG_AUTH_NOT_SUPPORTED
The LDAP server does not support strong authentication.

LDAP_STRONG_AUTH_REQUIRED
Strong authentication is required for the operation.

LDAP_PARTIAL_RESULTS
Partial results only returned.

LDAP_NO_SUCH_ATTRIBUTE
The attribute type specified does not exist in the entry.

LDAP_UNDEFINED_TYPE
: The attribute type specified is invalid.

LDAP_INAPPROPRIATE_MATCHING
: Filter type not supported for the specified attribute.

LDAP_CONSTRAINT_VIOLATION
: An attribute value specified violates some constraint (e.g., a postalAddress has too many lines, or a line that is too long).

LDAP_TYPE_OR_VALUE_EXISTS
: An attribute type or attribute value specified already exists in the entry.

LDAP_INVALID_SYNTAX
: An invalid attribute value was specified.

LDAP_NO_SUCH_OBJECT
: The specified object does not exist in The Directory.

LDAP_ALIAS_PROBLEM
: An alias in The Directory points to a nonexistent entry.

LDAP_INVALID_DN_SYNTAX
: A syntactically invalid DN was specified.

LDAP_IS_LEAF
: The object specified is a leaf.

LDAP_ALIAS_DEREF_PROBLEM
: A problem was encountered when dereferencing an alias.

LDAP_INAPPROPRIATE_AUTH
: Inappropriate authentication was specified (e.g., LDAP_AUTH_SIMPLE was specified and the entry does not have a userPassword attribute).

LDAP_INVALID_CREDENTIALS
: Invalid credentials were presented (e.g., the wrong password).

LDAP_INSUFFICIENT_ACCESS
: The user has insufficient access to perform the operation.

LDAP_BUSY
: The DSA is busy.

LDAP_UNAVAILABLE
: The DSA is unavailable.

LDAP_UNWILLING_TO_PERFORM
: The DSA is unwilling to perform the operation.

LDAP_LOOP_DETECT
: A loop was detected.

LDAP_NAMING_VIOLATION
: A naming violation occurred.

LDAP_OBJECT_CLASS_VIOLATION
: An object class violation occurred (e.g., a "must" attribute was missing from the entry).

LDAP_NOT_ALLOWED_ON_NONLEAF
: The operation is not allowed on a nonleaf object.

LDAP_NOT_ALLOWED_ON_RDN
: The operation is not allowed on an RDN.

LDAP_ALREADY_EXISTS
: The entry already exists.

LDAP_NO_OBJECT_CLASS_MODS
: Object class modifications are not allowed.

LDAP_OTHER An unknown error occurred.

API ERROR CODES

This section provides a complete list of API error codes recognized by the library. Note that LDAP_SUCCESS indicates success of an API call in addition to representing the return of the LDAP ´success' resultCode.

LDAP_SERVER_DOWN The LDAP library can't contact the LDAP server.

LDAP_LOCAL_ERROR Some local error occurred. This is usually a failed dynamic memory allocation.

LDAP_ENCODING_ERROR
An error was encountered encoding parameters to send to the LDAP server.

LDAP_DECODING_ERROR
An error was encountered decoding a result from the LDAP server.

LDAP_TIMEOUT A timelimit was exceeded while waiting for a result.

LDAP_AUTH_UNKNOWN
The authentication method specified to ldap_bind() is not known.

LDAP_FILTER_ERROR An invalid filter was supplied to ldap_search() (e.g., unbalanced parentheses).

LDAP_PARAM_ERROR An ldap routine was called with a bad parameter.

LDAP_NO_MEMORY An memory allocation (e.g., malloc(3) or other dynamic memory allocator) call failed in an ldap library routine.

LDAP_USER_CANCELED
Indicates the user cancelled the operation.

LDAP_CONNECT_ERROR
Indicates a connection problem.

LDAP_NOT_SUPPORTED
Indicates the routine was called in a manner not supported by the library.

LDAP_CONTROL_NOT_FOUND
Indicates the control provided is unknown to the client library.

LDAP_NO_RESULTS_RETURNED
Indicates no results returned.

LDAP_MORE_RESULTS_TO_RETURN
Indicates more results could be returned.

LDAP_CLIENT_LOOP Indicates the library has detected a loop in its processing.

LDAP_REFERRAL_LIMIT_EXCEEDED
Indicates the referral limit has been exceeded.

DEPRECATED

Deprecated interfaces generally remain in the library. The macro LDAP_DEPRECATED can be defined to a non-zero value (e.g., -DLDAP_DEPRECATED=1) when compiling program designed to use deprecated interfaces. It is recommended that developers writing new programs, or updating old programs, avoid use of deprecated interfaces. Over time, it is expected that documentation (and, eventually, support) for deprecated interfaces to be eliminated.

SEE ALSO

ldap(3),

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_extended_operation, ldap_extended_operation_s − Extends the LDAP operations to the LDAP server.

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_extended_operation(
LDAP ****ld*,**
const char ****requestoid*,**
const struct berval ****requestdata*,**
LDAPControl *****sctrls*,**
LDAPControl *****cctrls*,**
int ****msgidp* **);

int ldap_extended_operation_s(
LDAP ****ld*,**
const char ****requestoid*,**
const struct berval ****requestdata*,**
LDAPControl *****sctrls*,**
LDAPControl *****cctrls*,**
char *****retoidp*,**
struct berval *****retdatap* **);

DESCRIPTION

The **ldap_extended_operation_s()** routine is used to synchronously perform an LDAP extended operation. It takes *requestoid*, which points to a dotted-decimal OID string identifying the extended operation to perform. *requestdata* is the data required for the request, *sctrls* is an array of LDAPControl structures to use with this extended operation, *cctrls* is an array of LDAPControl structures that list the client controls to use with this extended operation.

The output parameter *retoidp* points to a dotted-decimal OID string returned by the LDAP server. The memory used by the string should be freed with the **ldap_memfree**(3) function. The output parameter *retdatap* points to a pointer to a berval structure that contains the returned data. If no data is returned by the server, the pointer is set this to NULL. The memory used by this structure should be freed with the **ber_bvfree**(3) function.

The **ldap_extended_operation()** works just like **ldap_extended_operation_s()**, but the operation is asynchronous. It provides the message id of the request it initiated in the integer pointed to be *msgidp*. The result of this operation can be obtained by calling **ldap_result(3).**

SEE ALSO

ber_bvfree(3), **ldap_memfree**(3), **ldap_parse_extended_result**(3), **ldap_result**(3)

ACKNOWLEDGEMENTS

NAME

ldap_extended_operation, ldap_extended_operation_s − Extends the LDAP operations to the LDAP server.

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_extended_operation(
LDAP ****ld*,**
const char ****requestoid*,**
const struct berval ****requestdata*,**
LDAPControl *****sctrls*,**
LDAPControl *****cctrls*,**
int ****msgidp* **);

int ldap_extended_operation_s(
LDAP ****ld*,**
const char ****requestoid*,**
const struct berval ****requestdata*,**
LDAPControl *****sctrls*,**
LDAPControl *****cctrls*,**
char *****retoidp*,**
struct berval *****retdatap* **);

DESCRIPTION

The **ldap_extended_operation_s()** routine is used to synchronously perform an LDAP extended operation. It takes *requestoid*, which points to a dotted-decimal OID string identifying the extended operation to perform. *requestdata* is the data required for the request, *sctrls* is an array of LDAPControl structures to use with this extended operation, *cctrls* is an array of LDAPControl structures that list the client controls to use with this extended operation.

The output parameter *retoidp* points to a dotted-decimal OID string returned by the LDAP server. The memory used by the string should be freed with the **ldap_memfree**(3) function. The output parameter *retdatap* points to a pointer to a berval structure that contains the returned data. If no data is returned by the server, the pointer is set this to NULL. The memory used by this structure should be freed with the **ber_bvfree**(3) function.

The **ldap_extended_operation()** works just like **ldap_extended_operation_s()**, but the operation is asynchronous. It provides the message id of the request it initiated in the integer pointed to be *msgidp*. The result of this operation can be obtained by calling **ldap_result(3).**

SEE ALSO

ber_bvfree(3), **ldap_memfree**(3), **ldap_parse_extended_result**(3), **ldap_result**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_first_attribute, ldap_next_attribute − step through LDAP entry attributes

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

char *ldap_first_attribute(
LDAP *ld, LDAPMessage *entry, BerElement **berptr)

char *ldap_next_attribute(
LDAP *ld, LDAPMessage *entry, BerElement *ber)

DESCRIPTION

The **ldap_first_attribute()** and **ldap_next_attribute()** routines are used to step through the attributes in an LDAP entry. **ldap_first_attribute()** takes an *entry* as returned by **ldap_first_entry**(3) or **ldap_next_entry**(3) and returns a pointer to character string containing the first attribute description in the entry. **ldap_next_attribute()** returns the next attribute description in the entry.

It also returns, in *berptr*, a pointer to a BerElement it has allocated to keep track of its current position. This pointer should be passed to subsequent calls to **ldap_next_attribute()** and is used to effectively step through the entry's attributes. The caller is solely responsible for freeing the BerElement pointed to by *berptr* when it is no longer needed by calling **ber_free**(3). When calling **ber_free**(3) in this instance, be sure the second argument is 0.

The attribute names returned are suitable for inclusion in a call to **ldap_get_values**(3) to retrieve the attribute's values.

ERRORS

If an error occurs, NULL is returned and the ld_errno field in the *ld* parameter is set to indicate the error. See **ldap_error**(3) for a description of possible error codes.

NOTES

The **ldap_first_attribute()** and **ldap_next_attribute()** return dynamically allocated memory that must be freed by the caller via **ldap_memfree**(3).

SEE ALSO

ldap(3), **ldap_first_entry**(3), **ldap_get_values**(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

NAME

ldap_first_attribute, ldap_next_attribute − step through LDAP entry attributes

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

char *ldap_first_attribute(
LDAP *ld, LDAPMessage *entry, BerElement **berptr)

char *ldap_next_attribute(
LDAP *ld, LDAPMessage *entry, BerElement *ber)

DESCRIPTION

The **ldap_first_attribute()** and **ldap_next_attribute()** routines are used to step through the attributes in an LDAP entry. **ldap_first_attribute()** takes an *entry* as returned by **ldap_first_entry**(3) or **ldap_next_entry**(3) and returns a pointer to character string containing the first attribute description in the entry. **ldap_next_attribute()** returns the next attribute description in the entry.

It also returns, in *berptr*, a pointer to a BerElement it has allocated to keep track of its current position. This pointer should be passed to subsequent calls to **ldap_next_attribute()** and is used to effectively step through the entry's attributes. The caller is solely responsible for freeing the BerElement pointed to by *berptr* when it is no longer needed by calling **ber_free**(3). When calling **ber_free**(3) in this instance, be sure the second argument is 0.

The attribute names returned are suitable for inclusion in a call to **ldap_get_values**(3) to retrieve the attribute's values.

ERRORS

If an error occurs, NULL is returned and the ld_errno field in the *ld* parameter is set to indicate the error. See **ldap_error**(3) for a description of possible error codes.

NOTES

The **ldap_first_attribute()** and **ldap_next_attribute()** return dynamically allocated memory that must be freed by the caller via **ldap_memfree**(3).

SEE ALSO

ldap(3), **ldap_first_entry**(3), **ldap_get_values**(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_first_entry, ldap_next_entry, ldap_count_entries − LDAP result entry parsing and counting routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_count_entries(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_first_entry(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_next_entry(LDAP *ld, LDAPMessage *entry)

DESCRIPTION

These routines are used to parse results received from **ldap_result**(3) or the synchronous LDAP search operation routines **ldap_search_s**(3) and **ldap_search_st**(3).

The **ldap_first_entry()** routine is used to retrieve the first entry in a chain of search results. It takes the *result* as returned by a call to **ldap_result**(3) or **ldap_search_s**(3) or **ldap_search_st**(3) and returns a pointer to the first entry in the result.

This pointer should be supplied on a subsequent call to **ldap_next_entry()** to get the next entry, the result of which should be supplied to the next call to **ldap_next_entry()**, etc. **ldap_next_entry()** will return NULL when there are no more entries. The entries returned from these calls are used in calls to the routines described in **ldap_get_dn**(3), **ldap_first_attribute**(3), **ldap_get_values**(3), etc.

A count of the number of entries in the search result can be obtained by calling **ldap_count_entries()**.

ERRORS

If an error occurs in **ldap_first_entry()** or **ldap_next_entry()**, NULL is returned and the ld_errno field in the *ld* parameter is set to indicate the error. If an error occurs in **ldap_count_entries()**, -1 is returned, and **ld_errno** is set appropriately. See **ldap_error**(3) for a description of possible error codes.

SEE ALSO

ldap(3), **ldap_result**(3), **ldap_search**(3), **ldap_first_attribute**(3), **ldap_get_values**(3), **ldap_get_dn**(3)

ACKNOWLEDGEMENTS

NAME

ldap_first_entry, ldap_next_entry, ldap_count_entries − LDAP result entry parsing and counting routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_count_entries(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_first_entry(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_next_entry(LDAP *ld, LDAPMessage *entry)

DESCRIPTION

These routines are used to parse results received from **ldap_result**(3) or the synchronous LDAP search operation routines **ldap_search_s**(3) and **ldap_search_st**(3).

The **ldap_first_entry()** routine is used to retrieve the first entry in a chain of search results. It takes the *result* as returned by a call to **ldap_result**(3) or **ldap_search_s**(3) or **ldap_search_st**(3) and returns a pointer to the first entry in the result.

This pointer should be supplied on a subsequent call to **ldap_next_entry()** to get the next entry, the result of which should be supplied to the next call to **ldap_next_entry()**, etc. **ldap_next_entry()** will return NULL when there are no more entries. The entries returned from these calls are used in calls to the routines described in **ldap_get_dn**(3), **ldap_first_attribute**(3), **ldap_get_values**(3), etc.

A count of the number of entries in the search result can be obtained by calling **ldap_count_entries()**.

ERRORS

If an error occurs in **ldap_first_entry()** or **ldap_next_entry()**, NULL is returned and the ld_errno field in the *ld* parameter is set to indicate the error. If an error occurs in **ldap_count_entries()**, -1 is returned, and **ld_errno** is set appropriately. See **ldap_error**(3) for a description of possible error codes.

SEE ALSO

ldap(3), **ldap_result**(3), **ldap_search**(3), **ldap_first_attribute**(3), **ldap_get_values**(3), **ldap_get_dn**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_first_message, ldap_next_message, ldap_count_messages − Stepping through messages in a result chain

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_count_messages(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_first_message(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_next_message(LDAP *ld, LDAPMessage *message)

DESCRIPTION

These routines are used to step through the messages in a result chain received from **ldap_result**(3)**.** For search operations, the result chain can contain referral, entry and result messages. The **ldap_msgtype**(3) function can be used to distinguish between the different message types.

The **ldap_first_message()** routine is used to retrieve the first message in a result chain. It takes the *result* as returned by a call to **ldap_result**(3)**, ldap_search_s**(3) or **ldap_search_st**(3) and returns a pointer to the first message in the result chain.

This pointer should be supplied on a subsequent call to **ldap_next_message()** to get the next message, the result of which should be supplied to the next call to **ldap_next_message()**, etc. **ldap_next_message()** will return NULL when there are no more messages.

These functions are useful when using routines like **ldap_parse_result**(3) that only operate on the first result in the chain.

A count of the number of messages in the result chain can be obtained by calling **ldap_count_messages()**. It can also be used to count the number of remaining messages in a chain if called with a message, entry or reference returned by **ldap_first_message() , ldap_next_message() , ldap_first_entry**(3)**, ldap_next_entry**(3)**, ldap_first_reference**(3)**, ldap_next_reference**(3)**.**

ERRORS

If an error occurs in **ldap_first_message()** or **ldap_next_message()**, NULL is returned. If an error occurs in **ldap_count_messages()**, -1 is returned.

SEE ALSO

ldap(3), **ldap_search**(3), **ldap_result**(3), **ldap_parse_result**(3), **ldap_first_entry**(3), **ldap_first_reference**(3)

ACKNOWLEDGEMENTS

NAME

ldap_first_message, ldap_next_message, ldap_count_messages − Stepping through messages in a result chain

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_count_messages(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_first_message(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_next_message(LDAP *ld, LDAPMessage *message)

DESCRIPTION

These routines are used to step through the messages in a result chain received from **ldap_result**(3)**.** For search operations, the result chain can contain referral, entry and result messages. The **ldap_msgtype**(3) function can be used to distinguish between the different message types.

The **ldap_first_message()** routine is used to retrieve the first message in a result chain. It takes the *result* as returned by a call to **ldap_result**(3)**, ldap_search_s**(3) or **ldap_search_st**(3) and returns a pointer to the first message in the result chain.

This pointer should be supplied on a subsequent call to **ldap_next_message()** to get the next message, the result of which should be supplied to the next call to **ldap_next_message()**, etc. **ldap_next_message()** will return NULL when there are no more messages.

These functions are useful when using routines like **ldap_parse_result**(3) that only operate on the first result in the chain.

A count of the number of messages in the result chain can be obtained by calling **ldap_count_messages()**. It can also be used to count the number of remaining messages in a chain if called with a message, entry or reference returned by **ldap_first_message() , ldap_next_message() , ldap_first_entry**(3)**, ldap_next_entry**(3)**, ldap_first_reference**(3)**, ldap_next_reference**(3)**.**

ERRORS

If an error occurs in **ldap_first_message()** or **ldap_next_message()**, NULL is returned. If an error occurs in **ldap_count_messages()**, -1 is returned.

SEE ALSO

ldap(3), **ldap_search**(3), **ldap_result**(3), **ldap_parse_result**(3), **ldap_first_entry**(3), **ldap_first_reference**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_first_reference, ldap_next_reference, ldap_count_references − Stepping through continuation references in a result chain

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_count_references(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_first_reference(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_next_reference(LDAP *ld, LDAPMessage *reference)

DESCRIPTION

These routines are used to step through the continuation references in a result chain received from **ldap_result**(3) or the synchronous LDAP search operation routines.

The **ldap_first_reference()** routine is used to retrieve the first reference message in a result chain. It takes the *result* as returned by a call to **ldap_result**(3)**, ldap_search_s**(3) or **ldap_search_st**(3) and returns a pointer to the first reference message in the result chain.

This pointer should be supplied on a subsequent call to **ldap_next_reference()** to get the next reference message, the result of which should be supplied to the next call to **ldap_next_reference()**, etc. **ldap_next_reference()** will return NULL when there are no more reference messages. The reference messages returned from these calls are used by **ldap_parse_reference**(3) to extract referrals and controls.

A count of the number of reference messages in the search result can be obtained by calling **ldap_count_references()**. It can also be used to count the number of reference messages remaining in a result chain.

ERRORS

If an error occurs in **ldap_first_reference()** or **ldap_next_reference()**, NULL is returned. If an error occurs in **ldap_count_references()**, -1 is returned.

SEE ALSO

ldap(3), **ldap_result**(3), **ldap_search**(3), **ldap_parse_reference**(3)

ACKNOWLEDGEMENTS

NAME

ldap_first_reference, ldap_next_reference, ldap_count_references − Stepping through continuation references in a result chain

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_count_references(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_first_reference(LDAP *ld, LDAPMessage *result)

LDAPMessage *ldap_next_reference(LDAP *ld, LDAPMessage *reference)

DESCRIPTION

These routines are used to step through the continuation references in a result chain received from **ldap_result**(3) or the synchronous LDAP search operation routines.

The **ldap_first_reference()** routine is used to retrieve the first reference message in a result chain. It takes the *result* as returned by a call to **ldap_result**(3)**, ldap_search_s**(3) or **ldap_search_st**(3) and returns a pointer to the first reference message in the result chain.

This pointer should be supplied on a subsequent call to **ldap_next_reference()** to get the next reference message, the result of which should be supplied to the next call to **ldap_next_reference()**, etc. **ldap_next_reference()** will return NULL when there are no more reference messages. The reference messages returned from these calls are used by **ldap_parse_reference**(3) to extract referrals and controls.

A count of the number of reference messages in the search result can be obtained by calling **ldap_count_references()**. It can also be used to count the number of reference messages remaining in a result chain.

ERRORS

If an error occurs in **ldap_first_reference()** or **ldap_next_reference()**, NULL is returned. If an error occurs in **ldap_count_references()**, -1 is returned.

SEE ALSO

ldap(3), **ldap_result**(3), **ldap_search**(3), **ldap_parse_reference**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_get_dn, ldap_explode_dn, ldap_explode_rdn, ldap_dn2ufn − LDAP DN handling routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

char *ldap_get_dn(LDAP *ld, LDAPMessage *entry)

int ldap_str2dn(const char *str, LDAPDN **dn, unsigned flags)

int ldap_dn2str(LDAPDN *dn, char **str, unsigned flags)

char **ldap_explode_dn(const char *dn, int notypes)

char **ldap_explode_rdn(const char *rdn, int notypes)

char *ldap_dn2ufn(const char * dn)

char *ldap_dn2dcedn(const char * dn)

char *ldap_dcedn2dn(const char * dn)

char *ldap_dn2ad_canonical(const char * dn)

DESCRIPTION

These routines allow LDAP entry names (Distinguished Names, or DNs) to be obtained, parsed, converted to a user-friendly form, and tested. A DN has the form described in RFC 4414 "Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names".

The **ldap_get_dn()** routine takes an *entry* as returned by **ldap_first_entry**(3) or **ldap_next_entry**(3) and returns a copy of the entry's DN. Space for the DN will be obtained dynamically and should be freed by the caller using **ldap_memfree**(3).

ldap_str2dn() parses a string representation of a distinguished name contained in **str** into its components, which are stored in **dn** as **ldap_ava** structures, arranged in **LDAPAVA, LDAPRDN,** and **LDAPDN** terms, defined as:

```
typedef struct ldap_ava {
   char *la_attr;
   struct berval *la_value;
   unsigned la_flags;
} LDAPAVA;

typedef LDAPAVA** LDAPRDN;
typedef LDAPRDN** LDAPDN;
```

The attribute types and the attribute values are not normalized. The **la_flags** can be either **LDAP_AVA_STRING** or **LDAP_AVA_BINARY,** the latter meaning that the value is BER/DER encoded and thus must be represented as, quoting from RFC 4514, " ... an octothorpe character ('#' ASCII 35) followed by the hexadecimal representation of each of the bytes of the BER encoding of the X.500 AttributeValue." The **flags** parameter to **ldap_str2dn()** can be

LDAP_DN_FORMAT_LDAPV3
LDAP_DN_FORMAT_LDAPV2
LDAP_DN_FORMAT_DCE

which defines what DN syntax is expected (according to RFC 4514, RFC 1779 and DCE, respectively). The format can be *OR*ed to the flags

LDAP_DN_P_NO_SPACES
LDAP_DN_P_NO_SPACE_AFTER_RDN

...
LDAP_DN_PEDANTIC

The latter is a shortcut for all the previous limitations.

LDAP_DN_P_NO_SPACES does not allow extra spaces in the dn; the default is to silently eliminate spaces around AVA separators ('='), RDN component separators ('+' for LDAPv3/LDAPv2 or ',' for DCE) and RDN separators (',' LDAPv3/LDAPv2 or '/' for DCE).

LDAP_DN_P_NO_SPACE_AFTER_RDN does not allow a single space after RDN separators.

ldap_dn2str() performs the inverse operation, yielding in **str** a string representation of **dn.** It allows the same values for **flags** as **ldap_str2dn(),** plus

LDAP_DN_FORMAT_UFN
LDAP_DN_FORMAT_AD_CANONICAL

for user-friendly naming (RFC 1781) and AD canonical.

The following routines are viewed as deprecated in favor of **ldap_str2dn()** and **ldap_dn2str().** They are provided to support legacy applications.

The **ldap_explode_dn()** routine takes a DN as returned by **ldap_get_dn()** and breaks it up into its component parts. Each part is known as a Relative Distinguished Name, or RDN. **ldap_explode_dn()** returns a NULL-terminated array, each component of which contains an RDN from the DN. The *notypes* parameter is used to request that only the RDN values be returned, not their types. For example, the DN "cn=Bob, c=US" would return as either { "cn=Bob", "c=US", NULL } or { "Bob", "US", NULL }, depending on whether notypes was 0 or 1, respectively. Assertion values in RDN strings may included escaped characters. The result can be freed by calling **ldap_value_free**(3).

Similarly, the **ldap_explode_rdn()** routine takes an RDN as returned by **ldap_explode_dn(dn,0)** and breaks it up into its "type=value" component parts (or just "value", if the *notypes* parameter is set). Note the value is not unescaped. The result can be freed by calling **ldap_value_free**(3).

ldap_dn2ufn() is used to turn a DN as returned by **ldap_get_dn**(3) into a more user-friendly form, stripping off all type names. See "Using the Directory to Achieve User Friendly Naming" (RFC 1781) for more details on the UFN format. Due to the ambiguous nature of the format, it is generally only used for display purposes. The space for the UFN returned is obtained dynamically and the user is responsible for freeing it via a call to **ldap_memfree**(3).

ldap_dn2dcedn() is used to turn a DN as returned by **ldap_get_dn**(3) into a DCE-style DN, e.g. a string with most-significant to least significant rdns separated by slashes ('/'); rdn components are separated by commas (','). Only printable chars (e.g. LDAPv2 printable string) are allowed, at least in this implementation. **ldap_dcedn2dn()** performs the opposite operation. **ldap_dn2ad_canonical()** turns a DN into a AD canonical name, which is basically a DCE dn with attribute types omitted. The trailing domain, if present, is turned in a DNS-like domain. The space for the returned value is obtained dynamically and the user is responsible for freeing it via a call to **ldap_memfree**(3).

ERRORS

If an error occurs in **ldap_get_dn()**, NULL is returned and the **ld_errno** field in the *ld* parameter is set to indicate the error. See **ldap_error**(3) for a description of possible error codes. **ldap_explode_dn()**, **ldap_explode_rdn()**, **ldap_dn2ufn(), ldap_dn2dcedn(), ldap_dcedn2dn(),** and **ldap_dn2ad_canonical()** will return NULL with **errno**(3) set appropriately in case of trouble.

NOTES

These routines dynamically allocate memory that the caller must free.

SEE ALSO

ldap(3), **ldap_error**(3), **ldap_first_entry**(3), **ldap_memfree**(3), **ldap_value_free**(3)

ACKNOWLEDGEMENTS

NAME

ldap_get_dn, ldap_explode_dn, ldap_explode_rdn, ldap_dn2ufn − LDAP DN handling routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

char *ldap_get_dn(LDAP *ld, LDAPMessage *entry)

int ldap_str2dn(const char *str, LDAPDN **dn, unsigned flags)

int ldap_dn2str(LDAPDN *dn, char **str, unsigned flags)

char **ldap_explode_dn(const char *dn, int notypes)

char **ldap_explode_rdn(const char *rdn, int notypes)

char *ldap_dn2ufn(const char * dn)

char *ldap_dn2dcedn(const char * dn)

char *ldap_dcedn2dn(const char * dn)

char *ldap_dn2ad_canonical(const char * dn)

DESCRIPTION

These routines allow LDAP entry names (Distinguished Names, or DNs) to be obtained, parsed, converted to a user-friendly form, and tested. A DN has the form described in RFC 4414 "Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names".

The **ldap_get_dn()** routine takes an *entry* as returned by **ldap_first_entry**(3) or **ldap_next_entry**(3) and returns a copy of the entry's DN. Space for the DN will be obtained dynamically and should be freed by the caller using **ldap_memfree**(3).

ldap_str2dn() parses a string representation of a distinguished name contained in **str** into its components, which are stored in **dn** as **ldap_ava** structures, arranged in **LDAPAVA, LDAPRDN,** and **LDAPDN** terms, defined as:

```
typedef struct ldap_ava {
   char *la_attr;
   struct berval *la_value;
   unsigned la_flags;
} LDAPAVA;

typedef LDAPAVA** LDAPRDN;
typedef LDAPRDN** LDAPDN;
```

The attribute types and the attribute values are not normalized. The **la_flags** can be either **LDAP_AVA_STRING** or **LDAP_AVA_BINARY,** the latter meaning that the value is BER/DER encoded and thus must be represented as, quoting from RFC 4514, " ... an octothorpe character ('#' ASCII 35) followed by the hexadecimal representation of each of the bytes of the BER encoding of the X.500 AttributeValue." The **flags** parameter to **ldap_str2dn()** can be

LDAP_DN_FORMAT_LDAPV3
LDAP_DN_FORMAT_LDAPV2
LDAP_DN_FORMAT_DCE

which defines what DN syntax is expected (according to RFC 4514, RFC 1779 and DCE, respectively). The format can be *OR*ed to the flags

LDAP_DN_P_NO_SPACES
LDAP_DN_P_NO_SPACE_AFTER_RDN

...
LDAP_DN_PEDANTIC

The latter is a shortcut for all the previous limitations.

LDAP_DN_P_NO_SPACES does not allow extra spaces in the dn; the default is to silently eliminate spaces around AVA separators ('='), RDN component separators ('+' for LDAPv3/LDAPv2 or ',' for DCE) and RDN separators (',' LDAPv3/LDAPv2 or '/' for DCE).

LDAP_DN_P_NO_SPACE_AFTER_RDN does not allow a single space after RDN separators.

ldap_dn2str() performs the inverse operation, yielding in **str** a string representation of **dn.** It allows the same values for **flags** as **ldap_str2dn(),** plus

LDAP_DN_FORMAT_UFN
LDAP_DN_FORMAT_AD_CANONICAL

for user-friendly naming (RFC 1781) and AD canonical.

The following routines are viewed as deprecated in favor of **ldap_str2dn()** and **ldap_dn2str().** They are provided to support legacy applications.

The **ldap_explode_dn()** routine takes a DN as returned by **ldap_get_dn()** and breaks it up into its component parts. Each part is known as a Relative Distinguished Name, or RDN. **ldap_explode_dn()** returns a NULL-terminated array, each component of which contains an RDN from the DN. The *notypes* parameter is used to request that only the RDN values be returned, not their types. For example, the DN "cn=Bob, c=US" would return as either { "cn=Bob", "c=US", NULL } or { "Bob", "US", NULL }, depending on whether notypes was 0 or 1, respectively. Assertion values in RDN strings may included escaped characters. The result can be freed by calling **ldap_value_free**(3).

Similarly, the **ldap_explode_rdn()** routine takes an RDN as returned by **ldap_explode_dn(dn,0)** and breaks it up into its "type=value" component parts (or just "value", if the *notypes* parameter is set). Note the value is not unescaped. The result can be freed by calling **ldap_value_free**(3).

ldap_dn2ufn() is used to turn a DN as returned by **ldap_get_dn**(3) into a more user-friendly form, stripping off all type names. See "Using the Directory to Achieve User Friendly Naming" (RFC 1781) for more details on the UFN format. Due to the ambiguous nature of the format, it is generally only used for display purposes. The space for the UFN returned is obtained dynamically and the user is responsible for freeing it via a call to **ldap_memfree**(3).

ldap_dn2dcedn() is used to turn a DN as returned by **ldap_get_dn**(3) into a DCE-style DN, e.g. a string with most-significant to least significant rdns separated by slashes ('/'); rdn components are separated by commas (','). Only printable chars (e.g. LDAPv2 printable string) are allowed, at least in this implementation. **ldap_dcedn2dn()** performs the opposite operation. **ldap_dn2ad_canonical()** turns a DN into a AD canonical name, which is basically a DCE dn with attribute types omitted. The trailing domain, if present, is turned in a DNS-like domain. The space for the returned value is obtained dynamically and the user is responsible for freeing it via a call to **ldap_memfree**(3).

ERRORS

If an error occurs in **ldap_get_dn()**, NULL is returned and the **ld_errno** field in the *ld* parameter is set to indicate the error. See **ldap_error**(3) for a description of possible error codes. **ldap_explode_dn()**, **ldap_explode_rdn()**, **ldap_dn2ufn(), ldap_dn2dcedn(), ldap_dcedn2dn(),** and **ldap_dn2ad_canonical()** will return NULL with **errno**(3) set appropriately in case of trouble.

NOTES

These routines dynamically allocate memory that the caller must free.

SEE ALSO

ldap(3), **ldap_error**(3), **ldap_first_entry**(3), **ldap_memfree**(3), **ldap_value_free**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_get_option, ldap_set_option − LDAP option handling routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_get_option(LDAP *_ld_, int _option_, void *_outvalue_);

int ldap_set_option(LDAP *_ld_, int _option_, const void *_invalue_);

DESCRIPTION

These routines provide access to options stored either in a LDAP handle or as global options, where applicable. They make use of a neutral interface, where the type of the value either retrieved by **ldap_get_option**(3) or set by **ldap_set_option**(3) is cast to **void *.** The actual type is determined based on the value of the **option** argument. Global options are set/retrieved by passing a NULL LDAP handle.

LDAP_OPT_API_INFO

Fills-in a **struct ldapapiinfo**; **outvalue** must be a **struct ldapapiinfo ***, pointing to an already allocated struct. This is a read-only option.

LDAP_OPT_DESC

Returns the file descriptor associated to the socket buffer of the LDAP handle passed in as **ld**; **outvalue** must be a **int *.** This is a read-only, handler-specific option.

LDAP_OPT_SOCKBUF

Returns a pointer to the socket buffer of the LDAP handle passed in as **ld**; **outvalue** must be a **Sockbuf **.** This is a read-only, handler-specific option.

LDAP_OPT_TIMEOUT

Sets/gets a timeout value for the synchronous API calls. **outvalue** must be a **struct timeval **** (the caller has to free ***outvalue**)**,** and **invalue** must be a **struct timeval ***, and they cannot be NULL. Using a struct with seconds set to -1 results in an infinite timeout, which is the default.

LDAP_OPT_NETWORK_TIMEOUT

Sets/gets the network timeout value after which **poll**(2)/**select**(2) following a **connect**(2) returns in case of no activity. **outvalue** must be a **struct timeval **** (the caller has to free ***outvalue**)**,** and **invalue** must be a **struct timeval ***, and they cannot be NULL. Using a struct with seconds set to -1 results in an infinite timeout, which is the default.

LDAP_OPT_DEREF

Sets/gets the value that defines when alias dereferencing must occur. **outvalue** and **invalue** must be **int ***, and they cannot be NULL.

LDAP_OPT_SIZELIMIT

Sets/gets the value that defines the maximum number of entries to be returned by a search operation. **outvalue** and **invalue** must be **int ***, and they cannot be NULL.

LDAP_OPT_TIMELIMIT

Sets/gets the value that defines the time limit after which a search operation should be terminated by the server. **outvalue** and **invalue** must be **int ***, and they cannot be NULL.

LDAP_OPT_REFERRALS

Determines whether the library should implicitly chase referrals or not. **outvalue** and **invalue** must be **int ***; their value should either be **LDAP_OPT_OFF** or **LDAP_OPT_ON**.

LDAP_OPT_RESTART

Determines whether the library should implicitly restart connections (FIXME). **outvalue** and **invalue** must be **int ***; their value should either be **LDAP_OPT_OFF** or **LDAP_OPT_ON**.

LDAP_OPT_PROTOCOL_VERSION
Sets/gets the protocol version. **outvalue** and **invalue** must be **int ***.

LDAP_OPT_SERVER_CONTROLS
Sets/gets the server-side controls to be used for all operations. This is now deprecated as modern LDAP C API provides replacements for all main operations which accepts server-side controls as explicit arguments; see for example **ldap_search_ext**(3), **ldap_add_ext**(3), **ldap_modify_ext**(3) and so on. **outvalue** must be **LDAPControl ***, and the caller is responsible of freeing the returned controls, if any, by calling **ldap_controls_free**(3), while **invalue** must be **LDAPControl **; the library duplicates the controls passed via **invalue**.

LDAP_OPT_CLIENT_CONTROLS
Sets/gets the client-side controls to be used for all operations. This is now deprecated as modern LDAP C API provides replacements for all main operations which accepts client-side controls as explicit arguments; see for example **ldap_search_ext**(3), **ldap_add_ext**(3), **ldap_modify_ext**(3) and so on. **outvalue** must be **LDAPControl ***, and the caller is responsible of freeing the returned controls, if any, by calling **ldap_controls_free**(3), while **invalue** must be **LDAPControl **; the library duplicates the controls passed via **invalue**.

LDAP_OPT_HOST_NAME
Sets/gets a space-separated list of hosts to be contacted by the library when trying to establish a connection. This is now deprecated in favor of **LDAP_OPT_URI**. **outvalue** must be a **char ****, and the caller is responsible of freeing the resulting string by calling **ldap_memfree**(3), while **invalue** must be a **char ***; the library duplicates the corresponding string.

LDAP_OPT_URI
Sets/gets a space-separated list of URIs to be contacted by the library when trying to establish a connection. **outvalue** must be a **char ****, and the caller is responsible of freeing the resulting string by calling **ldap_memfree**(3), while **invalue** must be a **char ***; the library parses the string into a list of **LDAPURLDesc** structures, so the invocation of **ldap_set_option**(3) may fail if URL parsing fails.

LDAP_OPT_DEFBASE
Sets/gets a string containing the DN to be used as default base for search operations. **outvalue** must be a **char ****, and the caller is responsible of freeing the returned string by calling **ldap_memfree**(3), while **invalue** must be a **char ***; the library duplicates the corresponding string.

LDAP_OPT_RESULT_CODE
Sets/gets the LDAP result code associated to the handle. This option was formerly known as **LDAP_OPT_ERROR_NUMBER**. Both **outvalue** and **invalue** must be a **int ***.

LDAP_OPT_DIAGNOSTIC_MESSAGE
Sets/gets a string containing the error string associated to the LDAP handle. This option was formerly known as **LDAP_OPT_ERROR_STRING**. **outvalue** must be a **char ****, and the caller is responsible of freeing the returned string by calling **ldap_memfree**(3), while **invalue** must be a **char ***; the library duplicates the corresponding string.

LDAP_OPT_MATCHED_DN
Sets/gets a string containing the matched DN associated to the LDAP handle. **outvalue** must be a **char ****, and the caller is responsible of freeing the returned string by calling **ldap_memfree**(3), while **invalue** must be a **char ***; the library duplicates the corresponding string.

LDAP_OPT_REFERRAL_URLS
Sets/gets an array containing the referral URIs associated to the LDAP handle. **outvalue** must be a **char *****, and the caller is responsible of freeing the returned string by calling **ber_memvfree**(3), while **invalue** must be a NULL-terminated **char ****; the library duplicates the corresponding string.

LDAP_OPT_API_FEATURE_INFO
Fills-in a **LDAPAPIFeatureInfo**; **outvalue** must be a **LDAPAPIFeatureInfo ***, pointing to an already allocated struct. This is a read-only option.

LDAP_OPT_DEBUG_LEVEL
Sets/gets the debug level of the client library. Both **outvalue** and **invalue** must be a **int ***.

ERRORS

On success, the functions return **LDAP_OPT_SUCCESS**, while they may return **LDAP_OPT_ERROR** to indicate a generic option handling error. Occasionally, more specific errors can be returned, like **LDAP_NO_MEMORY** to indicate a failure in memory allocation.

NOTES

The LDAP libraries with the **LDAP_OPT_REFERRALS** option set to **LDAP_OPT_ON** (default value) automatically follow referrals using an anonymous bind. Application developers are encouraged to either implement consistent referral chasing features, or explicitly disable referral chasing by setting that option to **LDAP_OPT_OFF**.

SEE ALSO

ldap(3), **ldap_error**(3), **RFC 4422** (http://www.rfc-editor.org),

ACKNOWLEDGEMENTS

NAME

ldap_get_option, ldap_set_option − LDAP option handling routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_get_option(LDAP *_ld_, int _option_, void *_outvalue_);

int ldap_set_option(LDAP *_ld_, int _option_, const void *_invalue_);

DESCRIPTION

These routines provide access to options stored either in a LDAP handle or as global options, where applicable. They make use of a neutral interface, where the type of the value either retrieved by **ldap_get_option**(3) or set by **ldap_set_option**(3) is cast to **void *.** The actual type is determined based on the value of the **option** argument. Global options are set/retrieved by passing a NULL LDAP handle.

LDAP_OPT_API_INFO

Fills-in a **struct ldapapiinfo**; **outvalue** must be a **struct ldapapiinfo ***, pointing to an already allocated struct. This is a read-only option.

LDAP_OPT_DESC

Returns the file descriptor associated to the socket buffer of the LDAP handle passed in as **ld**; **outvalue** must be a **int *.** This is a read-only, handler-specific option.

LDAP_OPT_SOCKBUF

Returns a pointer to the socket buffer of the LDAP handle passed in as **ld**; **outvalue** must be a **Sockbuf **.** This is a read-only, handler-specific option.

LDAP_OPT_TIMEOUT

Sets/gets a timeout value for the synchronous API calls. **outvalue** must be a **struct timeval **** (the caller has to free ***outvalue**)**,** and **invalue** must be a **struct timeval ***, and they cannot be NULL. Using a struct with seconds set to -1 results in an infinite timeout, which is the default.

LDAP_OPT_NETWORK_TIMEOUT

Sets/gets the network timeout value after which **poll**(2)/**select**(2) following a **connect**(2) returns in case of no activity. **outvalue** must be a **struct timeval **** (the caller has to free ***outvalue**)**,** and **invalue** must be a **struct timeval ***, and they cannot be NULL. Using a struct with seconds set to -1 results in an infinite timeout, which is the default.

LDAP_OPT_DEREF

Sets/gets the value that defines when alias dereferencing must occur. **outvalue** and **invalue** must be **int ***, and they cannot be NULL.

LDAP_OPT_SIZELIMIT

Sets/gets the value that defines the maximum number of entries to be returned by a search operation. **outvalue** and **invalue** must be **int ***, and they cannot be NULL.

LDAP_OPT_TIMELIMIT

Sets/gets the value that defines the time limit after which a search operation should be terminated by the server. **outvalue** and **invalue** must be **int ***, and they cannot be NULL.

LDAP_OPT_REFERRALS

Determines whether the library should implicitly chase referrals or not. **outvalue** and **invalue** must be **int ***; their value should either be **LDAP_OPT_OFF** or **LDAP_OPT_ON**.

LDAP_OPT_RESTART

Determines whether the library should implicitly restart connections (FIXME). **outvalue** and **invalue** must be **int ***; their value should either be **LDAP_OPT_OFF** or **LDAP_OPT_ON**.

LDAP_OPT_PROTOCOL_VERSION
Sets/gets the protocol version. **outvalue** and **invalue** must be **int ***.

LDAP_OPT_SERVER_CONTROLS
Sets/gets the server-side controls to be used for all operations. This is now deprecated as modern LDAP C API provides replacements for all main operations which accepts server-side controls as explicit arguments; see for example **ldap_search_ext**(3), **ldap_add_ext**(3), **ldap_modify_ext**(3) and so on. **outvalue** must be **LDAPControl *****, and the caller is responsible of freeing the returned controls, if any, by calling **ldap_controls_free**(3), while **invalue** must be **LDAPControl ****; the library duplicates the controls passed via **invalue**.

LDAP_OPT_CLIENT_CONTROLS
Sets/gets the client-side controls to be used for all operations. This is now deprecated as modern LDAP C API provides replacements for all main operations which accepts client-side controls as explicit arguments; see for example **ldap_search_ext**(3), **ldap_add_ext**(3), **ldap_modify_ext**(3) and so on. **outvalue** must be **LDAPControl *****, and the caller is responsible of freeing the returned controls, if any, by calling **ldap_controls_free**(3), while **invalue** must be **LDAPControl ****; the library duplicates the controls passed via **invalue**.

LDAP_OPT_HOST_NAME
Sets/gets a space-separated list of hosts to be contacted by the library when trying to establish a connection. This is now deprecated in favor of **LDAP_OPT_URI**. **outvalue** must be a **char ****, and the caller is responsible of freeing the resulting string by calling **ldap_memfree**(3), while **invalue** must be a **char ***; the library duplicates the corresponding string.

LDAP_OPT_URI
Sets/gets a space-separated list of URIs to be contacted by the library when trying to establish a connection. **outvalue** must be a **char ****, and the caller is responsible of freeing the resulting string by calling **ldap_memfree**(3), while **invalue** must be a **char ***; the library parses the string into a list of **LDAPURLDesc** structures, so the invocation of **ldap_set_option**(3) may fail if URL parsing fails.

LDAP_OPT_DEFBASE
Sets/gets a string containing the DN to be used as default base for search operations. **outvalue** must be a **char ****, and the caller is responsible of freeing the returned string by calling **ldap_memfree**(3), while **invalue** must be a **char ***; the library duplicates the corresponding string.

LDAP_OPT_RESULT_CODE
Sets/gets the LDAP result code associated to the handle. This option was formerly known as **LDAP_OPT_ERROR_NUMBER**. Both **outvalue** and **invalue** must be a **int ***.

LDAP_OPT_DIAGNOSTIC_MESSAGE
Sets/gets a string containing the error string associated to the LDAP handle. This option was formerly known as **LDAP_OPT_ERROR_STRING**. **outvalue** must be a **char ****, and the caller is responsible of freeing the returned string by calling **ldap_memfree**(3), while **invalue** must be a **char ***; the library duplicates the corresponding string.

LDAP_OPT_MATCHED_DN
Sets/gets a string containing the matched DN associated to the LDAP handle. **outvalue** must be a **char ****, and the caller is responsible of freeing the returned string by calling **ldap_memfree**(3), while **invalue** must be a **char ***; the library duplicates the corresponding string.

LDAP_OPT_REFERRAL_URLS
Sets/gets an array containing the referral URIs associated to the LDAP handle. **outvalue** must be a **char *****, and the caller is responsible of freeing the returned string by calling **ber_memvfree**(3), while **invalue** must be a NULL-terminated **char ****; the library duplicates the corresponding string.

LDAP_OPT_API_FEATURE_INFO
Fills-in a **LDAPAPIFeatureInfo**; **outvalue** must be a **LDAPAPIFeatureInfo ***, pointing to an already allocated struct. This is a read-only option.

LDAP_OPT_DEBUG_LEVEL
Sets/gets the debug level of the client library. Both **outvalue** and **invalue** must be a **int ***.

ERRORS

On success, the functions return **LDAP_OPT_SUCCESS**, while they may return **LDAP_OPT_ERROR** to indicate a generic option handling error. Occasionally, more specific errors can be returned, like **LDAP_NO_MEMORY** to indicate a failure in memory allocation.

NOTES

The LDAP libraries with the **LDAP_OPT_REFERRALS** option set to **LDAP_OPT_ON** (default value) automatically follow referrals using an anonymous bind. Application developers are encouraged to either implement consistent referral chasing features, or explicitly disable referral chasing by setting that option to **LDAP_OPT_OFF**.

SEE ALSO

ldap(3), **ldap_error**(3), **RFC 4422** (http://www.rfc-editor.org),

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_get_values, ldap_get_values_len, ldap_count_values − LDAP attribute value handling routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

char **ldap_get_values(ld, entry, attr)
LDAP *ld;
LDAPMessage *entry;
char *attr;

struct berval **ldap_get_values_len(ld, entry, attr)
LDAP *ld;
LDAPMessage *entry;
char *attr;

int ldap_count_values(vals)
char **vals;

int ldap_count_values_len(vals)
struct berval **vals;

void ldap_value_free(vals)
char **vals;

void ldap_value_free_len(vals)
struct berval **vals;

DESCRIPTION

These routines are used to retrieve and manipulate attribute values from an LDAP entry as returned by **ldap_first_entry**(3) or **ldap_next_entry**(3). **ldap_get_values()** takes the *entry* and the attribute *attr* whose values are desired and returns a NULL-terminated array of the attribute's values. *attr* may be an attribute type as returned from **ldap_first_attribute**(3) or **ldap_next_attribute**(3), or if the attribute type is known it can simply be given.

The number of values in the array can be counted by calling **ldap_count_values()**. The array of values returned can be freed by calling **ldap_value_free()**.

If the attribute values are binary in nature, and thus not suitable to be returned as an array of char *'s, the **ldap_get_values_len()** routine can be used instead. It takes the same parameters as **ldap_get_values()**, but returns a NULL-terminated array of pointers to berval structures, each containing the length of and a pointer to a value.

The number of values in the array can be counted by calling **ldap_count_values_len()**. The array of values returned can be freed by calling **ldap_value_free_len()**.

ERRORS

If an error occurs in **ldap_get_values()** or **ldap_get_values_len()**, NULL is returned and the **ld_errno** field in the *ld* parameter is set to indicate the error. See **ldap_error**(3) for a description of possible error codes.

NOTES

These routines dynamically allocate memory which the caller must free using the supplied routines.

SEE ALSO

ldap(3), **ldap_first_entry**(3), **ldap_first_attribute**(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

NAME

ldap_get_values, ldap_get_values_len, ldap_count_values − LDAP attribute value handling routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

char **ldap_get_values(ld, entry, attr)
LDAP *ld;
LDAPMessage *entry;
char *attr;

struct berval **ldap_get_values_len(ld, entry, attr)
LDAP *ld;
LDAPMessage *entry;
char *attr;

int ldap_count_values(vals)
char **vals;

int ldap_count_values_len(vals)
struct berval **vals;

void ldap_value_free(vals)
char **vals;

void ldap_value_free_len(vals)
struct berval **vals;

DESCRIPTION

These routines are used to retrieve and manipulate attribute values from an LDAP entry as returned by **ldap_first_entry**(3) or **ldap_next_entry**(3). **ldap_get_values()** takes the *entry* and the attribute *attr* whose values are desired and returns a NULL-terminated array of the attribute's values. *attr* may be an attribute type as returned from **ldap_first_attribute**(3) or **ldap_next_attribute**(3), or if the attribute type is known it can simply be given.

The number of values in the array can be counted by calling **ldap_count_values()**. The array of values returned can be freed by calling **ldap_value_free()**.

If the attribute values are binary in nature, and thus not suitable to be returned as an array of char *'s, the **ldap_get_values_len()** routine can be used instead. It takes the same parameters as **ldap_get_values()**, but returns a NULL-terminated array of pointers to berval structures, each containing the length of and a pointer to a value.

The number of values in the array can be counted by calling **ldap_count_values_len()**. The array of values returned can be freed by calling **ldap_value_free_len()**.

ERRORS

If an error occurs in **ldap_get_values()** or **ldap_get_values_len()**, NULL is returned and the **ld_errno** field in the *ld* parameter is set to indicate the error. See **ldap_error**(3) for a description of possible error codes.

NOTES

These routines dynamically allocate memory which the caller must free using the supplied routines.

SEE ALSO

ldap(3), **ldap_first_entry**(3), **ldap_first_attribute**(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_memfree, ldap_memvfree, ldap_memalloc, ldap_memcalloc, ldap_memrealloc, ldap_strdup − LDAP memory allocation routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

void ldap_memfree(void * *p***);**

void ldap_memvfree(void ** *v***);**

void *ldap_memalloc(ber_len_t *s***);**

void *ldap_memcalloc(ber_len_t *n***, ber_len_t** *s***);**

void *ldap_memrealloc(void * *p***, ber_len_t** *s***);**

char *ldap_strdup(LDAP_CONST char * *p***);**

DESCRIPTION

These routines are used to allocate/deallocate memory used/returned by the LDAP library. **ldap_memalloc**(), **ldap_memcalloc**(), **ldap_memrealloc**(), and **ldap_memfree**() are used exactly like the standard **malloc**(3), **calloc**(3), **realloc**(3), and **free**(3) routines, respectively. The **ldap_memvfree**() routine is used to free a dynamically allocated array of pointers to arbitrary dynamically allocated objects. The **ldap_strdup**() routine is used exactly like the standard **strdup**(3) routine.

SEE ALSO

ldap(3)

ACKNOWLEDGEMENTS

NAME

ldap_memfree, ldap_memvfree, ldap_memalloc, ldap_memcalloc, ldap_memrealloc, ldap_strdup − LDAP memory allocation routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

void ldap_memfree(void * *p***);**

void ldap_memvfree(void ** *v***);**

void *ldap_memalloc(ber_len_t *s***);**

void *ldap_memcalloc(ber_len_t *n***, ber_len_t** *s***);**

void *ldap_memrealloc(void * *p***, ber_len_t** *s***);**

char *ldap_strdup(LDAP_CONST char * *p***);**

DESCRIPTION

These routines are used to allocate/deallocate memory used/returned by the LDAP library. **ldap_memalloc**(), **ldap_memcalloc**(), **ldap_memrealloc**(), and **ldap_memfree**() are used exactly like the standard **malloc**(3), **calloc**(3), **realloc**(3), and **free**(3) routines, respectively. The **ldap_memvfree**() routine is used to free a dynamically allocated array of pointers to arbitrary dynamically allocated objects. The **ldap_strdup**() routine is used exactly like the standard **strdup**(3) routine.

SEE ALSO

ldap(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_modify_ext, ldap_modify_ext_s − Perform an LDAP modify operation

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_modify_ext(
LDAP ****ld*,**
char ****dn*,**
LDAPMod ****mods[]*,**
LDAPControl *****sctrls*,**
LDAPControl *****cctrls*,**
int *****msgidp* **);

int ldap_modify_ext_s(
LDAP ****ld*,**
char ****dn*,**
LDAPMod ****mods[]*,**
LDAPControl *****sctrls*,**
LDAPControl *****cctrls* **);

void ldap_mods_free(
LDAPMod *****mods*,**
int *freemods* **);**

DESCRIPTION

The routine **ldap_modify_ext_s()** is used to perform an LDAP modify operation. *dn* is the DN of the entry to modify, and *mods* is a null-terminated array of modifications to make to the entry. Each element of the *mods* array is a pointer to an LDAPMod structure, which is defined below.

```
typedef struct ldapmod {
   int mod_op;
   char *mod_type;
   union {
      char **modv_strvals;
      struct berval **modv_bvals;
   } mod_vals;
   struct ldapmod *mod_next;
} LDAPMod;
#define mod_values mod_vals.modv_strvals
#define mod_bvalues mod_vals.modv_bvals
```

The *mod_op* field is used to specify the type of modification to perform and should be one of LDAP_MOD_ADD, LDAP_MOD_DELETE, or LDAP_MOD_REPLACE. The *mod_type* and *mod_values* fields specify the attribute type to modify and a null-terminated array of values to add, delete, or replace respectively. The *mod_next* field is used only by the LDAP server and may be ignored by the client.

If you need to specify a non-string value (e.g., to add a photo or audio attribute value), you should set *mod_op* to the logical OR of the operation as above (e.g., LDAP_MOD_REPLACE) and the constant LDAP_MOD_BVALUES. In this case, *mod_bvalues* should be used instead of *mod_values*, and it should point to a null-terminated array of struct bervals, as defined in <lber.h>.

For LDAP_MOD_ADD modifications, the given values are added to the entry, creating the attribute if necessary. For LDAP_MOD_DELETE modifications, the given values are deleted from the entry, removing the attribute if no values remain. If the entire attribute is to be deleted, the *mod_values* field should be set to NULL. For LDAP_MOD_REPLACE modifications, the attribute will have the listed values after the modification, having been created if necessary. All modifications are performed in the order in which they

are listed.

ldap_mods_free() can be used to free each element of a NULL-terminated array of mod structures. If *freemods* is non-zero, the *mods* pointer itself is freed as well.

ldap_modify_ext_s() returns a code indicating success or, in the case of failure, indicating the nature of the failure. See **ldap_error**(3) for details

The **ldap_modify_ext()** operation works the same way as **ldap_modify_ext_s()**, except that it is asynchronous. The integer that *msgidp* points to is set to the message id of the modify request. The result of the operation can be obtained by calling **ldap_result**(3).

Both **ldap_modify_ext()** and **ldap_modify_ext_s()** allows server and client controls to be passed in via the sctrls and cctrls parameters, respectively.

DEPRECATED INTERFACES

The **ldap_modify()** and **ldap_modify_s()** routines are deprecated in favor of the **ldap_modify_ext()** and **ldap_modify_ext_s()** routines, respectively.

Deprecated interfaces generally remain in the library. The macro LDAP_DEPRECATED can be defined to a non-zero value (e.g., -DLDAP_DEPRECATED=1) when compiling program designed to use deprecated interfaces. It is recommended that developers writing new programs, or updating old programs, avoid use of deprecated interfaces. Over time, it is expected that documentation (and, eventually, support) for deprecated interfaces to be eliminated.

SEE ALSO

ldap(3), **ldap_error**(3),

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_modrdn, ldap_modrdn_s, ldap_modrdn2, ldap_modrdn2_s − Perform an LDAP modify RDN operation

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_modrdn(ld, dn, newrdn)
LDAP ∗ld;
char ∗dn, ∗newrdn;

int ldap_modrdn_s(ld, dn, newrdn)
LDAP ∗ld;
char ∗dn, ∗newrdn;

int ldap_modrdn2(ld, dn, newrdn, deleteoldrdn)
LDAP ∗ld;
char ∗dn, ∗newrdn;
int deleteoldrdn;

int ldap_modrdn2_s(ld, dn, newrdn, deleteoldrdn)
LDAP ∗ld;
char ∗dn, ∗newrdn;
int deleteoldrdn;

DESCRIPTION

The **ldap_modrdn()** and **ldap_modrdn_s()** routines perform an LDAP modify RDN operation. They both take *dn*, the DN of the entry whose RDN is to be changed, and *newrdn*, the new RDN to give the entry. The old RDN of the entry is never kept as an attribute of the entry. **ldap_modrdn()** is asynchronous, returning the message id of the operation it initiates. **ldap_modrdn_s()** is synchronous, returning the LDAP error code indicating the success or failure of the operation. Use of these routines is deprecated. Use the versions described below instead.

The **ldap_modrdn2()** and **ldap_modrdn2_s()** routines also perform an LDAP modify RDN operation, taking the same parameters as above. In addition, they both take the *deleteoldrdn* parameter which is used as a boolean value to indicate whether the old RDN values should be deleted from the entry or not.

ERRORS

The synchronous (_s) versions of these routines return an LDAP error code, either LDAP_SUCCESS or an error if there was trouble. The asynchronous versions return -1 in case of trouble, setting the **ld_errno** field of *ld*. See **ldap_error**(3) for more details.

SEE ALSO

ldap(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

NAME

ldap_modrdn, ldap_modrdn_s, ldap_modrdn2, ldap_modrdn2_s − Perform an LDAP modify RDN operation

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_modrdn(ld, dn, newrdn)
LDAP ∗ld;
char ∗dn, ∗newrdn;

int ldap_modrdn_s(ld, dn, newrdn)
LDAP ∗ld;
char ∗dn, ∗newrdn;

int ldap_modrdn2(ld, dn, newrdn, deleteoldrdn)
LDAP ∗ld;
char ∗dn, ∗newrdn;
int deleteoldrdn;

int ldap_modrdn2_s(ld, dn, newrdn, deleteoldrdn)
LDAP ∗ld;
char ∗dn, ∗newrdn;
int deleteoldrdn;

DESCRIPTION

The **ldap_modrdn()** and **ldap_modrdn_s()** routines perform an LDAP modify RDN operation. They both take *dn*, the DN of the entry whose RDN is to be changed, and *newrdn*, the new RDN to give the entry. The old RDN of the entry is never kept as an attribute of the entry. **ldap_modrdn()** is asynchronous, returning the message id of the operation it initiates. **ldap_modrdn_s()** is synchronous, returning the LDAP error code indicating the success or failure of the operation. Use of these routines is deprecated. Use the versions described below instead.

The **ldap_modrdn2()** and **ldap_modrdn2_s()** routines also perform an LDAP modify RDN operation, taking the same parameters as above. In addition, they both take the *deleteoldrdn* parameter which is used as a boolean value to indicate whether the old RDN values should be deleted from the entry or not.

ERRORS

The synchronous (_s) versions of these routines return an LDAP error code, either LDAP_SUCCESS or an error if there was trouble. The asynchronous versions return -1 in case of trouble, setting the **ld_errno** field of *ld*. See **ldap_error**(3) for more details.

SEE ALSO

ldap(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_init, ldap_initialize, ldap_open − Initialize the LDAP library and open a connection to an LDAP server

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

LDAP *ldap_open(host, port)
char *host;
int port;

LDAP *ldap_init(host, port)
char *host;
int port;

int ldap_initialize(ldp, uri)
LDAP **ldp;
char *uri;

#include <ldap_pvt.h>

int ldap_init_fd(fd, proto, uri, ldp)
ber_socket_t fd;
int proto;
char *uri;
LDAP **ldp;

DESCRIPTION

ldap_open() opens a connection to an LDAP server and allocates an LDAP structure which is used to identify the connection and to maintain per-connection information. **ldap_init()** allocates an LDAP structure but does not open an initial connection. **ldap_initialize()** allocates an LDAP structure but does not open an initial connection. **ldap_init_fd()** allocates an LDAP structure using an existing connection on the provided socket. One of these routines must be called before any operations are attempted.

ldap_open() takes *host*, the hostname on which the LDAP server is running, and *port*, the port number to which to connect. If the default IANA-assigned port of 389 is desired, LDAP_PORT should be specified for *port*. The *host* parameter may contain a blank-separated list of hosts to try to connect to, and each host may optionally by of the form *host:port*. If present, the *:port* overrides the *port* parameter to **ldap_open()**. Upon successfully making a connection to an LDAP server, **ldap_open()** returns a pointer to an opaque LDAP structure, which should be passed to subsequent calls to **ldap_bind()**, **ldap_search()**, etc. Certain fields in the LDAP structure can be set to indicate size limit, time limit, and how aliases are handled during operations; read and write access to those fields must occur by calling **ldap_get_option**(3) and **ldap_set_option**(3) respectively, whenever possible.

ldap_init() acts just like **ldap_open()**, but does not open a connection to the LDAP server. The actual connection open will occur when the first operation is attempted.

ldap_initialize() acts like **ldap_init()**, but it returns an integer indicating either success or the failure reason, and it allows to specify details for the connection in the schema portion of the URI.

At this time, **ldap_open()** and **ldap_init()** are deprecated in favor of **ldap_initialize()**, essentially because the latter allows to specify a schema in the URI and it explicitly returns an error code.

ldap_init_fd() allows an LDAP structure to be initialized using an already-opened connection. The *proto* parameter should be one of LDAP_PROTO_TCP, LDAP_PROTO_UDP, or LDAP_PROTO_IPC for a connection using TCP, UDP, or IPC, respectively. The value LDAP_PROTO_EXT may also be specified if user-supplied sockbuf handlers are going to be used. Note that support for UDP is not implemented unless libldap was built with LDAP_CONNECTIONLESS defined. The *uri* parameter may optionally be provided for informational purposes.

Note: the first call into the LDAP library also initializes the global options for the library. As such the first call should be single-threaded or otherwise protected to insure that only one call is active. It is recommended that **ldap_get_option**() or **ldap_set_option**() be used in the program's main thread before any additional threads are created. See **ldap_get_option**(3).

ERRORS

If an error occurs, **ldap_open()** and **ldap_init()** will return NULL and errno should be set appropriately. **ldap_initialize()** and **ldap_init_fd()** will directly return the LDAP code associated to the error (or *LDAP_SUCCESS* in case of success); errno should be set as well whenever appropriate.

SEE ALSO

ldap(3), **ldap_bind**(3), **ldap_get_option**(3), **ldap_set_option**(3), **lber-sockbuf**(3), **errno**(3)

ACKNOWLEDGEMENTS

NAME

ldap_init, ldap_initialize, ldap_open − Initialize the LDAP library and open a connection to an LDAP server

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

LDAP *ldap_open(host, port)
char *host;
int port;

LDAP *ldap_init(host, port)
char *host;
int port;

int ldap_initialize(ldp, uri)
LDAP **ldp;
char *uri;

#include <ldap_pvt.h>

int ldap_init_fd(fd, proto, uri, ldp)
ber_socket_t fd;
int proto;
char *uri;
LDAP **ldp;

DESCRIPTION

ldap_open() opens a connection to an LDAP server and allocates an LDAP structure which is used to identify the connection and to maintain per-connection information. **ldap_init()** allocates an LDAP structure but does not open an initial connection. **ldap_initialize()** allocates an LDAP structure but does not open an initial connection. **ldap_init_fd()** allocates an LDAP structure using an existing connection on the provided socket. One of these routines must be called before any operations are attempted.

ldap_open() takes *host*, the hostname on which the LDAP server is running, and *port*, the port number to which to connect. If the default IANA-assigned port of 389 is desired, LDAP_PORT should be specified for *port*. The *host* parameter may contain a blank-separated list of hosts to try to connect to, and each host may optionally by of the form *host:port*. If present, the *:port* overrides the *port* parameter to **ldap_open()**. Upon successfully making a connection to an LDAP server, **ldap_open()** returns a pointer to an opaque LDAP structure, which should be passed to subsequent calls to **ldap_bind()**, **ldap_search()**, etc. Certain fields in the LDAP structure can be set to indicate size limit, time limit, and how aliases are handled during operations; read and write access to those fields must occur by calling **ldap_get_option**(3) and **ldap_set_option**(3) respectively, whenever possible.

ldap_init() acts just like **ldap_open()**, but does not open a connection to the LDAP server. The actual connection open will occur when the first operation is attempted.

ldap_initialize() acts like **ldap_init()**, but it returns an integer indicating either success or the failure reason, and it allows to specify details for the connection in the schema portion of the URI.

At this time, **ldap_open()** and **ldap_init()** are deprecated in favor of **ldap_initialize()**, essentially because the latter allows to specify a schema in the URI and it explicitly returns an error code.

ldap_init_fd() allows an LDAP structure to be initialized using an already-opened connection. The *proto* parameter should be one of LDAP_PROTO_TCP, LDAP_PROTO_UDP, or LDAP_PROTO_IPC for a connection using TCP, UDP, or IPC, respectively. The value LDAP_PROTO_EXT may also be specified if user-supplied sockbuf handlers are going to be used. Note that support for UDP is not implemented unless libldap was built with LDAP_CONNECTIONLESS defined. The *uri* parameter may optionally be provided for informational purposes.

Note: the first call into the LDAP library also initializes the global options for the library. As such the first call should be single-threaded or otherwise protected to insure that only one call is active. It is recommended that **ldap_get_option**() or **ldap_set_option**() be used in the program's main thread before any additional threads are created. See **ldap_get_option**(3).

ERRORS

If an error occurs, **ldap_open()** and **ldap_init()** will return NULL and errno should be set appropriately. **ldap_initialize()** and **ldap_init_fd()** will directly return the LDAP code associated to the error (or *LDAP_SUCCESS* in case of success); errno should be set as well whenever appropriate.

SEE ALSO

ldap(3), **ldap_bind**(3), **ldap_get_option**(3), **ldap_set_option**(3), **lber-sockbuf**(3), **errno**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_parse_reference − Extract referrals and controls from a reference message

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_parse_reference(LDAP *ld, LDAPMessage *reference,
char ***referralsp, LDAPControl ***serverctrlsp,
int freeit)

DESCRIPTION

The **ldap_parse_reference()** routine is used to extract referrals and controls from a reference message. The *reference* parameter is a reference message as returned by a call to **ldap_first_reference**(3)**, ldap_next_reference**(3)**, ldap_first_message**(3)**, ldap_next_message**(3)**,** or **ldap_result**(3)**.**

The *referralsp* parameter will be filled in with an allocated array of character strings. The strings are copies of the referrals contained in the parsed message. The array should be freed by calling **ldap_value_free**(3)**.** If *referralsp* is NULL, no referrals are returned. If no referrals were returned, **referralsp* is set to NULL.

The *serverctrlsp* parameter will be filled in with an allocated array of controls copied from the parsed message. The array should be freed by calling **ldap_controls_free**(3). If *serverctrlsp* is NULL, no controls are returned. If no controls were returned, **serverctrlsp* is set to NULL.

The *freeit* parameter determines whether the parsed message is freed or not after the extraction. Any non-zero value will make it free the message. The **ldap_msgfree**(3) routine can also be used to free the message later.

ERRORS

Upon success LDAP_SUCCESS is returned. Otherwise the values of the *referralsp* and *serverctrlsp* parameters are undefined.

SEE ALSO

ldap(3), **ldap_first_reference**(3), **ldap_first_message**(3), **ldap_result**(3), **ldap_get_values**(3), **ldap_controls_free**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_parse_result − Parsing results

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_parse_result(LDAP *ld, LDAPMessage *result,
int *errcodep, char **matcheddnp, char **errmsgp,
char ***referralsp, LDAPControl ***serverctrlsp,
int freeit)

int ldap_parse_sasl_bind_result(LDAP *ld, LDAPMessage *result,
struct berval **servercredp, int freeit)

int ldap_parse_extended_result(LDAP *ld, LDAPMessage *result,
char **retoidp, struct berval **retdatap, int freeit)

DESCRIPTION

These routines are used to extract information from a result message. They will operate on the first result message in a chain of search results (skipping past other message types). They take the *result* as returned by a call to **ldap_result**(3), **ldap_search_s**(3) or **ldap_search_st**(3). In addition to **ldap_parse_result()**, the routines **ldap_parse_sasl_bind_result()** and **ldap_parse_extended_result()** are used to get all the result information from SASL bind and extended operations.

The *errcodep* parameter will be filled in with the result code from the result message.

The server might supply a matched DN string in the message indicating how much of a name in a request was recognized. The *matcheddnp* parameter will be filled in with this string if supplied, else it will be NULL. If a string is returned, it should be freed using **ldap_memfree**(3).

The *errmsgp* parameter will be filled in with the error message field from the parsed message. This string should be freed using **ldap_memfree**(3).

The *referralsp* parameter will be filled in with an allocated array of referral strings from the parsed message. This array should be freed using **ldap_memvfree**(3). If no referrals were returned, **referralsp* is set to NULL.

The *serverctrlsp* parameter will be filled in with an allocated array of controls copied from the parsed message. The array should be freed using **ldap_controls_free**(3). If no controls were returned, **serverctrlsp* is set to NULL.

The *freeit* parameter determines whether the parsed message is freed or not after the extraction. Any non-zero value will make it free the message. The **ldap_msgfree**(3) routine can also be used to free the message later.

For SASL bind results, the *servercredp* parameter will be filled in with an allocated berval structure containing the credentials from the server if present. The structure should be freed using **ber_bvfree**(3).

For extended results, the *retoidp* parameter will be filled in with the dotted-OID text representation of the name of the extended operation response. The string should be freed using **ldap_memfree**(3). If no OID was returned, **retoidp* is set to NULL.

For extended results, the *retdatap* parameter will be filled in with a pointer to a berval structure containing the data from the extended operation response. The structure should be freed using **ber_bvfree**(3). If no data were returned, **retdatap* is set to NULL.

For all the above result parameters, NULL values can be used in calls in order to ignore certain fields.

ERRORS

Upon success LDAP_SUCCESS is returned. Otherwise the values of the result parameters are undefined.

SEE ALSO

ldap(3), **ldap_result**(3), **ldap_search**(3), **ldap_memfree**(3), **ldap_memvfree**(3), **ldap_get_values**(3), **ldap_controls_free**(3), **lber-types**(3)

ACKNOWLEDGEMENTS

NAME

ldap_parse_result − Parsing results

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_parse_result(LDAP *ld, LDAPMessage *result,
int *errcodep, char **matcheddnp, char **errmsgp,
char ***referralsp, LDAPControl ***serverctrlsp,
int freeit)

int ldap_parse_sasl_bind_result(LDAP *ld, LDAPMessage *result,
struct berval **servercredp, int freeit)

int ldap_parse_extended_result(LDAP *ld, LDAPMessage *result,
char **retoidp, struct berval **retdatap, int freeit)

DESCRIPTION

These routines are used to extract information from a result message. They will operate on the first result message in a chain of search results (skipping past other message types). They take the *result* as returned by a call to **ldap_result**(3), **ldap_search_s**(3) or **ldap_search_st**(3). In addition to **ldap_parse_result()**, the routines **ldap_parse_sasl_bind_result()** and **ldap_parse_extended_result()** are used to get all the result information from SASL bind and extended operations.

The *errcodep* parameter will be filled in with the result code from the result message.

The server might supply a matched DN string in the message indicating how much of a name in a request was recognized. The *matcheddnp* parameter will be filled in with this string if supplied, else it will be NULL. If a string is returned, it should be freed using **ldap_memfree**(3).

The *errmsgp* parameter will be filled in with the error message field from the parsed message. This string should be freed using **ldap_memfree**(3).

The *referralsp* parameter will be filled in with an allocated array of referral strings from the parsed message. This array should be freed using **ldap_memvfree**(3). If no referrals were returned, **referralsp* is set to NULL.

The *serverctrlsp* parameter will be filled in with an allocated array of controls copied from the parsed message. The array should be freed using **ldap_controls_free**(3). If no controls were returned, **serverctrlsp* is set to NULL.

The *freeit* parameter determines whether the parsed message is freed or not after the extraction. Any non-zero value will make it free the message. The **ldap_msgfree**(3) routine can also be used to free the message later.

For SASL bind results, the *servercredp* parameter will be filled in with an allocated berval structure containing the credentials from the server if present. The structure should be freed using **ber_bvfree**(3).

For extended results, the *retoidp* parameter will be filled in with the dotted-OID text representation of the name of the extended operation response. The string should be freed using **ldap_memfree**(3). If no OID was returned, **retoidp* is set to NULL.

For extended results, the *retdatap* parameter will be filled in with a pointer to a berval structure containing the data from the extended operation response. The structure should be freed using **ber_bvfree**(3). If no data were returned, **retdatap* is set to NULL.

For all the above result parameters, NULL values can be used in calls in order to ignore certain fields.

ERRORS

Upon success LDAP_SUCCESS is returned. Otherwise the values of the result parameters are undefined.

SEE ALSO

ldap(3), **ldap_result**(3), **ldap_search**(3), **ldap_memfree**(3), **ldap_memvfree**(3), **ldap_get_values**(3), **ldap_controls_free**(3), **lber-types**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_parse_sort_control − Decode the information returned from a search operation that used a server-side sort control

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_parse_sort_control(ld, ctrls, returnCode, attribute)
LDAP *ld;
LDAPControl **ctrls;
unsigned long *returnCode;
char **attribute;

DESCRIPTION

This function is used to parse the results returned in a search operation that uses a server-side sort control.

It takes a null terminated array of LDAPControl structures usually obtained by a call to the **ldap_parse_result** function. A returncode which points to the sort control result code,and an array of LDAPControl structures that list the client controls to use with the search. The function also takes an out parameter *attribute* and if the sort operation fails, the server may return a string that indicates the first attribute in the sortKey list that caused the failure. If this parameter is NULL, no string is returned. If a string is returned, the memory should be freed by calling the ldap_memfree function.

NOTES

SEE ALSO

ldap_result(3), **ldap_controls_free**(3)

ACKNOWLEDGEMENTS

NAME

ldap_parse_sort_control − Decode the information returned from a search operation that used a server-side sort control

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_parse_sort_control(ld, ctrls, returnCode, attribute)
LDAP *ld;
LDAPControl **ctrls;
unsigned long *returnCode;
char **attribute;

DESCRIPTION

This function is used to parse the results returned in a search operation that uses a server-side sort control.

It takes a null terminated array of LDAPControl structures usually obtained by a call to the **ldap_parse_result** function. A returncode which points to the sort control result code,and an array of LDAPControl structures that list the client controls to use with the search. The function also takes an out parameter *attribute* and if the sort operation fails, the server may return a string that indicates the first attribute in the sortKey list that caused the failure. If this parameter is NULL, no string is returned. If a string is returned, the memory should be freed by calling the ldap_memfree function.

NOTES

SEE ALSO

ldap_result(3), **ldap_controls_free**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_parse_vlv_control − Decode the information returned from a search operation that used a VLV (virtual list view) control

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_parse_vlv_control(ld, ctrlp, target_posp, list_countp, contextp, errcodep)
LDAP *ld;
LDAPControl **ctrlp;
unsigned long *target_posp, *list_countp;
struct berval **contextp;
int *errcodep;

DESCRIPTION

The **ldap_parse_vlv_control** is used to decode the information returned from a search operation that used a VLV (virtual list view)control. It takes a null terminated array of LDAPControl structures, usually obtained by a call to the **ldap_parse_result**function, a *target_pos* which points to the list index of the target entry. If this parameter is NULL, the target position is not returned. The index returned is an approximation of the position of the target entry. It is not guaranteed to be exact. The parameter *list_countp* points to the server's estimate of the size of the list. If this parameter is NULL, the size is not returned. *contextp* is a pointer to the address of a berval structure that contains a server-generated context identifier if server returns one. If server does not return a context identifier, the server returns a NULL in this parameter. If this parameter is set to NULL, the context identifier is not returned. You should use this returned context in the next call to create a VLV control. When the berval structure is no longer needed, you should free the memory by calling the *ber_bvfree function.e errcodep* is an output parameter, which points to the result code returned by the server. If this parameter is NULL, the result code is not returned.

See ldap.h for a list of possible return codes.

SEE ALSO

ldap_search(3)

ACKNOWLEDGEMENTS

NAME

ldap_parse_vlv_control − Decode the information returned from a search operation that used a VLV (virtual list view) control

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_parse_vlv_control(ld, ctrlp, target_posp, list_countp, contextp, errcodep)
LDAP *ld;
LDAPControl **ctrlp;
unsigned long *target_posp, *list_countp;
struct berval **contextp;
int *errcodep;

DESCRIPTION

The **ldap_parse_vlv_control** is used to decode the information returned from a search operation that used a VLV (virtual list view)control. It takes a null terminated array of LDAPControl structures, usually obtained by a call to the **ldap_parse_result**function, a *target_pos* which points to the list index of the target entry. If this parameter is NULL, the target position is not returned. The index returned is an approximation of the position of the target entry. It is not guaranteed to be exact. The parameter *list_countp* points to the server's estimate of the size of the list. If this parameter is NULL, the size is not returned. *contextp* is a pointer to the address of a berval structure that contains a server-generated context identifier if server returns one. If server does not return a context identifier, the server returns a NULL in this parameter. If this parameter is set to NULL, the context identifier is not returned. You should use this returned context in the next call to create a VLV control. When the berval structure is no longer needed, you should free the memory by calling the *ber_bvfree function.e errcodep* is an output parameter, which points to the result code returned by the server. If this parameter is NULL, the result code is not returned.

See ldap.h for a list of possible return codes.

SEE ALSO

ldap_search(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_rename, ldap_rename_s − Renames the specified entry.

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_rename(ld, dn, newrdn, newparent, deleteoldrdn, sctrls[], cctrls[], msgidp);
LDAP *ld;
const char *dn, *newrdn, *newparent;
int deleteoldrdn;
LDAPControl *sctrls[], *cctrls[];
int *msgidp);

int ldap_rename_s(ld, dn, newrdn, newparent, deleteoldrdn, sctrls[], cctrls[]);
LDAP *ld;
const char *dn, *newrdn, *newparent;
int deleteoldrdn;
LDAPControl *sctrls[], *cctrls[];

DESCRIPTION

These routines are used to perform a LDAP rename operation. The function changes the leaf component of an entry's distinguished name and optionally moves the entry to a new parent container. The **ldap_rename_s** performs a rename operation synchronously. The method takes *dn*, which points to the distinguished name of the entry whose attribute is being compared, *newparent*,the distinguished name of the entry's new parent. If this parameter is NULL, only the RDN is changed. The root DN is specified by passing a zero length string, "". *deleteoldrdn* specifies whether the old RDN should be retained or deleted. Zero indicates that the old RDN should be retained. If you choose this option, the attribute will contain both names (the old and the new). Non-zero indicates that the old RDN should be deleted. *serverctrls* points to an array of LDAPControl structures that list the client controls to use with this extended operation. Use NULL to specify no client controls. *clientctrls* points to an array of LDAPControl structures that list the client controls to use with the search.

ldap_rename works just like **ldap_rename_s,** but the operation is asynchronous. It returns the message id of the request it initiated. The result of this operation can be obtained by calling **ldap_result(3).**

ERRORS

ldap_rename() returns -1 in case of error initiating the request, and will set the *ld_errno* field in the *ld* parameter to indicate the error. **ldap_rename_s()** returns the LDAP error code resulting from the rename operation.

SEE ALSO

ldap(3), **ldap_modify**(3)

ACKNOWLEDGEMENTS

NAME

ldap_rename, ldap_rename_s − Renames the specified entry.

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_rename(ld, dn, newrdn, newparent, deleteoldrdn, sctrls[], cctrls[], msgidp);
LDAP *ld;
const char *dn, *newrdn, *newparent;
int deleteoldrdn;
LDAPControl *sctrls[], *cctrls[];
int *msgidp);

int ldap_rename_s(ld, dn, newrdn, newparent, deleteoldrdn, sctrls[], cctrls[]);
LDAP *ld;
const char *dn, *newrdn, *newparent;
int deleteoldrdn;
LDAPControl *sctrls[], *cctrls[];

DESCRIPTION

These routines are used to perform a LDAP rename operation. The function changes the leaf component of an entry's distinguished name and optionally moves the entry to a new parent container. The **ldap_rename_s** performs a rename operation synchronously. The method takes *dn*, which points to the distinguished name of the entry whose attribute is being compared, *newparent*,the distinguished name of the entry's new parent. If this parameter is NULL, only the RDN is changed. The root DN is specified by passing a zero length string, "". *deleteoldrdn* specifies whether the old RDN should be retained or deleted. Zero indicates that the old RDN should be retained. If you choose this option, the attribute will contain both names (the old and the new). Non-zero indicates that the old RDN should be deleted. *serverctrls* points to an array of LDAPControl structures that list the client controls to use with this extended operation. Use NULL to specify no client controls. *clientctrls* points to an array of LDAPControl structures that list the client controls to use with the search.

ldap_rename works just like **ldap_rename_s,** but the operation is asynchronous. It returns the message id of the request it initiated. The result of this operation can be obtained by calling **ldap_result(3).**

ERRORS

ldap_rename() returns -1 in case of error initiating the request, and will set the *ld_errno* field in the *ld* parameter to indicate the error. **ldap_rename_s()** returns the LDAP error code resulting from the rename operation.

SEE ALSO

ldap(3), **ldap_modify**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_result − Wait for the result of an LDAP operation

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_result(LDAP *ld, int msgid, int all,
struct timeval *timeout, LDAPMessage **result);

int ldap_msgfree(LDAPMessage *msg);

int ldap_msgtype(LDAPMessage *msg);

int ldap_msgid(LDAPMessage *msg);

DESCRIPTION

The **ldap_result()** routine is used to wait for and return the result of an operation previously initiated by one of the LDAP asynchronous operation routines (e.g., **ldap_search_ext**(3), **ldap_modify_ext**(3), etc.). Those routines all return -1 in case of error, and an invocation identifier upon successful initiation of the operation. The invocation identifier is picked by the library and is guaranteed to be unique across the LDAP session. It can be used to request the result of a specific operation from **ldap_result()** through the *msgid* parameter.

The **ldap_result()** routine will block or not, depending upon the setting of the *timeout* parameter. If timeout is not a NULL pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a NULL pointer, the LDAP_OPT_TIMEOUT value set by **ldap_set_option**(3) is used. With the default setting, the select blocks indefinitely. To effect a poll, the timeout argument should be a non-NULL pointer, pointing to a zero-valued timeval structure. See **select**(2) for further details.

If the result of a specific operation is required, *msgid* should be set to the invocation identifier returned when the operation was initiated, otherwise LDAP_RES_ANY or LDAP_RES_UNSOLICITED should be supplied to wait for any or unsolicited response.

The *all* parameter, if non-zero, causes **ldap_result()** to return all responses with msgid, otherwise only the next response is returned. This is commonly used to obtain all the responses of a search operation.

A search response is made up of zero or more search entries, zero or more search references, and zero or more extended partial responses followed by a search result. If *all* is set to 0, search entries will be returned one at a time as they come in, via separate calls to **ldap_result()**. If it's set to 1, the search response will only be returned in its entirety, i.e., after all entries, all references, all extended partial responses, and the final search result have been received.

Upon success, the type of the result received is returned and the *result* parameter will contain the result of the operation; otherwise, the *result* parameter is undefined. This result should be passed to the LDAP parsing routines, **ldap_first_message**(3) and friends, for interpretation.

The possible result types returned are:

LDAP_RES_BIND (0x61)
LDAP_RES_SEARCH_ENTRY (0x64)
LDAP_RES_SEARCH_REFERENCE (0x73)
LDAP_RES_SEARCH_RESULT (0x65)
LDAP_RES_MODIFY (0x67)
LDAP_RES_ADD (0x69)
LDAP_RES_DELETE (0x6b)
LDAP_RES_MODDN (0x6d)
LDAP_RES_COMPARE (0x6f)
LDAP_RES_EXTENDED (0x78)

LDAP_RES_INTERMEDIATE (0x79)

The **ldap_msgfree()** routine is used to free the memory allocated for result(s) by **ldap_result()** or **ldap_search_ext_s**(3) and friends. It takes a pointer to the result or result chain to be freed and returns the type of the last message in the chain. If the parameter is NULL, the function does nothing and returns zero.

The **ldap_msgtype()** routine returns the type of a message.

The **ldap_msgid()** routine returns the message id of a message.

ERRORS

ldap_result() returns -1 if something bad happens, and zero if the timeout specified was exceeded. **ldap_msgtype()** and **ldap_msgid()** return -1 on error.

SEE ALSO

ldap(3), **ldap_first_message**(3), **select**(2)

ACKNOWLEDGEMENTS

NAME

ldap_result − Wait for the result of an LDAP operation

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_result(LDAP *ld, int msgid, int all,
struct timeval *timeout, LDAPMessage **result);

int ldap_msgfree(LDAPMessage *msg);

int ldap_msgtype(LDAPMessage *msg);

int ldap_msgid(LDAPMessage *msg);

DESCRIPTION

The **ldap_result()** routine is used to wait for and return the result of an operation previously initiated by one of the LDAP asynchronous operation routines (e.g., **ldap_search_ext**(3), **ldap_modify_ext**(3), etc.). Those routines all return -1 in case of error, and an invocation identifier upon successful initiation of the operation. The invocation identifier is picked by the library and is guaranteed to be unique across the LDAP session. It can be used to request the result of a specific operation from **ldap_result()** through the *msgid* parameter.

The **ldap_result()** routine will block or not, depending upon the setting of the *timeout* parameter. If timeout is not a NULL pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a NULL pointer, the LDAP_OPT_TIMEOUT value set by **ldap_set_option**(3) is used. With the default setting, the select blocks indefinitely. To effect a poll, the timeout argument should be a non-NULL pointer, pointing to a zero-valued timeval structure. See **select**(2) for further details.

If the result of a specific operation is required, *msgid* should be set to the invocation identifier returned when the operation was initiated, otherwise LDAP_RES_ANY or LDAP_RES_UNSOLICITED should be supplied to wait for any or unsolicited response.

The *all* parameter, if non-zero, causes **ldap_result()** to return all responses with msgid, otherwise only the next response is returned. This is commonly used to obtain all the responses of a search operation.

A search response is made up of zero or more search entries, zero or more search references, and zero or more extended partial responses followed by a search result. If *all* is set to 0, search entries will be returned one at a time as they come in, via separate calls to **ldap_result()**. If it's set to 1, the search response will only be returned in its entirety, i.e., after all entries, all references, all extended partial responses, and the final search result have been received.

Upon success, the type of the result received is returned and the *result* parameter will contain the result of the operation; otherwise, the *result* parameter is undefined. This result should be passed to the LDAP parsing routines, **ldap_first_message**(3) and friends, for interpretation.

The possible result types returned are:

LDAP_RES_BIND (0x61)
LDAP_RES_SEARCH_ENTRY (0x64)
LDAP_RES_SEARCH_REFERENCE (0x73)
LDAP_RES_SEARCH_RESULT (0x65)
LDAP_RES_MODIFY (0x67)
LDAP_RES_ADD (0x69)
LDAP_RES_DELETE (0x6b)
LDAP_RES_MODDN (0x6d)
LDAP_RES_COMPARE (0x6f)
LDAP_RES_EXTENDED (0x78)

LDAP_RES_INTERMEDIATE (0x79)

The **ldap_msgfree()** routine is used to free the memory allocated for result(s) by **ldap_result()** or **ldap_search_ext_s**(3) and friends. It takes a pointer to the result or result chain to be freed and returns the type of the last message in the chain. If the parameter is NULL, the function does nothing and returns zero.

The **ldap_msgtype()** routine returns the type of a message.

The **ldap_msgid()** routine returns the message id of a message.

ERRORS

ldap_result() returns -1 if something bad happens, and zero if the timeout specified was exceeded. **ldap_msgtype()** and **ldap_msgid()** return -1 on error.

SEE ALSO

ldap(3), **ldap_first_message**(3), **select**(2)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_str2syntax, ldap_syntax2str, ldap_syntax2name, ldap_syntax_free, ldap_str2matchingrule, ldap_matchingrule2str, ldap_matchingrule2name, ldap_matchingrule_free, ldap_str2attributetype, ldap_attributetype2str, ldap_attributetype2name, ldap_attributetype_free, ldap_str2objectclass, ldap_objectclass2str, ldap_objectclass2name, ldap_objectclass_free, ldap_scherr2str − Schema definition handling routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>
#include <ldap_schema.h>

LDAPSyntax * ldap_str2syntax(s, code, errp, flags)
const char * s;
int * code;
const char ** errp;
const int flags;

char * ldap_syntax2str(syn)
const LDAPSyntax * syn;

const char * ldap_syntax2name(syn)
LDAPSyntax * syn;

ldap_syntax_free(syn)
LDAPSyntax * syn;

LDAPMatchingRule * ldap_str2matchingrule(s, code, errp, flags)
const char * s;
int * code;
const char ** errp;
const int flags;

char * ldap_matchingrule2str(mr);
const LDAPMatchingRule * mr;

const char * ldap_matchingrule2name(mr)
LDAPMatchingRule * mr;

ldap_matchingrule_free(mr)
LDAPMatchingRule * mr;

LDAPAttributeType * ldap_str2attributetype(s, code, errp, flags)
const char * s;
int * code;
const char ** errp;
const int flags;

char * ldap_attributetype2str(at)
const LDAPAttributeType * at;

const char * ldap_attributetype2name(at)
LDAPAttributeType * at;

ldap_attributetype_free(at)
LDAPAttributeType * at;

LDAPObjectClass * ldap_str2objectclass(s, code, errp, flags)
const char * s;
int * code;
const char ** errp;

const int flags;

char * ldap_objectclass2str(oc)
const LDAPObjectClass * oc;

const char * ldap_objectclass2name(oc)
LDAPObjectClass * oc;

ldap_objectclass_free(oc)
LDAPObjectClass * oc;

char * ldap_scherr2str(code)
int code;

DESCRIPTION

These routines are used to parse schema definitions in the syntax defined in RFC 4512 into structs and handle these structs. These routines handle four kinds of definitions: syntaxes, matching rules, attribute types and object classes. For each definition kind, four routines are provided.

ldap_str2xxx() takes a definition in RFC 4512 format in argument *s* as a NUL-terminated string and returns, if possible, a pointer to a newly allocated struct of the appropriate kind. The caller is responsible for freeing the struct by calling **ldap_xxx_free()** when not needed any longer. The routine returns NULL if some problem happened. In this case, the integer pointed at by argument *code* will receive an error code (see below the description of **ldap_scherr2str()** for an explanation of the values) and a pointer to a NUL-terminated string will be placed where requested by argument *errp* , indicating where in argument *s* the error happened, so it must not be freed by the caller. Argument *flags* is a bit mask of parsing options controlling the relaxation of the syntax recognized. The following values are defined:

LDAP_SCHEMA_ALLOW_NONE
strict parsing according to RFC 4512.

LDAP_SCHEMA_ALLOW_NO_OID
permit definitions that do not contain an initial OID.

LDAP_SCHEMA_ALLOW_QUOTED
permit quotes around some items that should not have them.

LDAP_SCHEMA_ALLOW_DESCR
permit a **descr** instead of a numeric OID in places where the syntax expect the latter.

LDAP_SCHEMA_ALLOW_DESCR_PREFIX
permit that the initial numeric OID contains a prefix in **descr** format.

LDAP_SCHEMA_ALLOW_ALL
be very liberal, include all options.

The structures returned are as follows:

```
typedef struct ldap_schema_extension_item {
        char *lsei_name;            /* Extension name */
        char **lsei_values;         /* Extension values */
} LDAPSchemaExtensionItem;

typedef struct ldap_syntax {
        char *syn_oid;              /* OID */
        char **syn_names;           /* Names */
        char *syn_desc;             /* Description */
        LDAPSchemaExtensionItem **syn_extensions; /* Extension */
} LDAPSyntax;

typedef struct ldap_matchingrule {
        char *mr_oid;               /* OID */
```

```
        char **mr_names;            /* Names */
        char *mr_desc;              /* Description */
        int  mr_obsolete;           /* Is obsolete? */
        char *mr_syntax_oid;        /* Syntax of asserted values */
        LDAPSchemaExtensionItem **mr_extensions; /* Extensions */
} LDAPMatchingRule;

typedef struct ldap_attributetype {
        char *at_oid;               /* OID */
        char **at_names;            /* Names */
        char *at_desc;              /* Description */
        int  at_obsolete;           /* Is obsolete? */
        char *at_sup_oid;           /* OID of superior type */
        char *at_equality_oid;      /* OID of equality matching rule */
        char *at_ordering_oid;      /* OID of ordering matching rule */
        char *at_substr_oid;        /* OID of substrings matching rule */
        char *at_syntax_oid;        /* OID of syntax of values */
        int  at_syntax_len;         /* Suggested minimum maximum length */
        int  at_single_value;       /* Is single-valued?  */
        int  at_collective;         /* Is collective? */
        int  at_no_user_mod;        /* Are changes forbidden through LDAP? */
        int  at_usage;              /* Usage, see below */
        LDAPSchemaExtensionItem **at_extensions; /* Extensions */
} LDAPAttributeType;

typedef struct ldap_objectclass {
        char *oc_oid;               /* OID */
        char **oc_names;            /* Names */
        char *oc_desc;              /* Description */
        int  oc_obsolete;           /* Is obsolete? */
        char **oc_sup_oids;         /* OIDs of superior classes */
        int  oc_kind;               /* Kind, see below */
        char **oc_at_oids_must;     /* OIDs of required attribute types */
        char **oc_at_oids_may;      /* OIDs of optional attribute types */
        LDAPSchemaExtensionItem **oc_extensions; /* Extensions */
} LDAPObjectClass;
```

Some integer fields (those described with a question mark) have a truth value, for these fields the possible values are:

LDAP_SCHEMA_NO
The answer to the question is no.

LDAP_SCHEMA_YES
The answer to the question is yes.

For attribute types, the following usages are possible:

LDAP_SCHEMA_USER_APPLICATIONS
the attribute type is non-operational.

LDAP_SCHEMA_DIRECTORY_OPERATION
the attribute type is operational and is pertinent to the directory itself, i.e. it has the same value on all servers that master the entry containing this attribute type.

LDAP_SCHEMA_DISTRIBUTED_OPERATION
the attribute type is operational and is pertinent to replication, shadowing or other distributed directory aspect. TBC.

LDAP_SCHEMA_DSA_OPERATION
the attribute type is operational and is pertinent to the directory server itself, i.e. it may have different values for the same entry when retrieved from different servers that master the entry.

Object classes can be of three kinds:

LDAP_SCHEMA_ABSTRACT
the object class is abstract, i.e. there cannot be entries of this class alone.

LDAP_SCHEMA_STRUCTURAL
the object class is structural, i.e. it describes the main role of the entry. On some servers, once the entry is created the set of structural object classes assigned cannot be changed: none of those present can be removed and none other can be added.

LDAP_SCHEMA_AUXILIARY
the object class is auxiliary, i.e. it is intended to go with other, structural, object classes. These can be added or removed at any time if attribute types are added or removed at the same time as needed by the set of object classes resulting from the operation.

Routines **ldap_xxx2name()** return a canonical name for the definition.

Routines **ldap_xxx2str()** return a string representation in the format described by RFC 4512 of the struct passed in the argument. The string is a newly allocated string that must be freed by the caller. These routines may return NULL if no memory can be allocated for the string.

ldap_scherr2str() returns a NUL-terminated string with a text description of the error found. This is a pointer to a static area, so it must not be freed by the caller. The argument *code* comes from one of the parsing routines and can adopt the following values:

LDAP_SCHERR_OUTOFMEM
Out of memory.

LDAP_SCHERR_UNEXPTOKEN
Unexpected token.

LDAP_SCHERR_NOLEFTPAREN
Missing opening parenthesis.

LDAP_SCHERR_NORIGHTPAREN
Missing closing parenthesis.

LDAP_SCHERR_NODIGIT
Expecting digit.

LDAP_SCHERR_BADNAME
Expecting a name.

LDAP_SCHERR_BADDESC
Bad description.

LDAP_SCHERR_BADSUP
Bad superiors.

LDAP_SCHERR_DUPOPT
Duplicate option.

LDAP_SCHERR_EMPTY
Unexpected end of data.

SEE ALSO

ldap(3)

ACKNOWLEDGEMENTS

NAME

ldap_str2syntax, ldap_syntax2str, ldap_syntax2name, ldap_syntax_free, ldap_str2matchingrule, ldap_matchingrule2str, ldap_matchingrule2name, ldap_matchingrule_free, ldap_str2attributetype, ldap_attributetype2str, ldap_attributetype2name, ldap_attributetype_free, ldap_str2objectclass, ldap_objectclass2str, ldap_objectclass2name, ldap_objectclass_free, ldap_scherr2str − Schema definition handling routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>
#include <ldap_schema.h>

LDAPSyntax * ldap_str2syntax(s, code, errp, flags)
const char * s;
int * code;
const char ** errp;
const int flags;

char * ldap_syntax2str(syn)
const LDAPSyntax * syn;

const char * ldap_syntax2name(syn)
LDAPSyntax * syn;

ldap_syntax_free(syn)
LDAPSyntax * syn;

LDAPMatchingRule * ldap_str2matchingrule(s, code, errp, flags)
const char * s;
int * code;
const char ** errp;
const int flags;

char * ldap_matchingrule2str(mr);
const LDAPMatchingRule * mr;

const char * ldap_matchingrule2name(mr)
LDAPMatchingRule * mr;

ldap_matchingrule_free(mr)
LDAPMatchingRule * mr;

LDAPAttributeType * ldap_str2attributetype(s, code, errp, flags)
const char * s;
int * code;
const char ** errp;
const int flags;

char * ldap_attributetype2str(at)
const LDAPAttributeType * at;

const char * ldap_attributetype2name(at)
LDAPAttributeType * at;

ldap_attributetype_free(at)
LDAPAttributeType * at;

LDAPObjectClass * ldap_str2objectclass(s, code, errp, flags)
const char * s;
int * code;
const char ** errp;

const int flags;

char * ldap_objectclass2str(oc)
const LDAPObjectClass * oc;

const char * ldap_objectclass2name(oc)
LDAPObjectClass * oc;

ldap_objectclass_free(oc)
LDAPObjectClass * oc;

char * ldap_scherr2str(code)
int code;

DESCRIPTION

These routines are used to parse schema definitions in the syntax defined in RFC 4512 into structs and handle these structs. These routines handle four kinds of definitions: syntaxes, matching rules, attribute types and object classes. For each definition kind, four routines are provided.

ldap_str2xxx() takes a definition in RFC 4512 format in argument *s* as a NUL-terminated string and returns, if possible, a pointer to a newly allocated struct of the appropriate kind. The caller is responsible for freeing the struct by calling **ldap_xxx_free()** when not needed any longer. The routine returns NULL if some problem happened. In this case, the integer pointed at by argument *code* will receive an error code (see below the description of **ldap_scherr2str()** for an explanation of the values) and a pointer to a NUL-terminated string will be placed where requested by argument *errp* , indicating where in argument *s* the error happened, so it must not be freed by the caller. Argument *flags* is a bit mask of parsing options controlling the relaxation of the syntax recognized. The following values are defined:

LDAP_SCHEMA_ALLOW_NONE
strict parsing according to RFC 4512.

LDAP_SCHEMA_ALLOW_NO_OID
permit definitions that do not contain an initial OID.

LDAP_SCHEMA_ALLOW_QUOTED
permit quotes around some items that should not have them.

LDAP_SCHEMA_ALLOW_DESCR
permit a **descr** instead of a numeric OID in places where the syntax expect the latter.

LDAP_SCHEMA_ALLOW_DESCR_PREFIX
permit that the initial numeric OID contains a prefix in **descr** format.

LDAP_SCHEMA_ALLOW_ALL
be very liberal, include all options.

The structures returned are as follows:

```
typedef struct ldap_schema_extension_item {
        char *lsei_name;            /* Extension name */
        char **lsei_values;         /* Extension values */
} LDAPSchemaExtensionItem;

typedef struct ldap_syntax {
        char *syn_oid;              /* OID */
        char **syn_names;           /* Names */
        char *syn_desc;             /* Description */
        LDAPSchemaExtensionItem **syn_extensions; /* Extension */
} LDAPSyntax;

typedef struct ldap_matchingrule {
        char *mr_oid;               /* OID */
```

```
        char **mr_names;            /* Names */
        char *mr_desc;              /* Description */
        int  mr_obsolete;           /* Is obsolete? */
        char *mr_syntax_oid;        /* Syntax of asserted values */
        LDAPSchemaExtensionItem **mr_extensions; /* Extensions */
} LDAPMatchingRule;

typedef struct ldap_attributetype {
        char *at_oid;               /* OID */
        char **at_names;            /* Names */
        char *at_desc;              /* Description */
        int  at_obsolete;           /* Is obsolete? */
        char *at_sup_oid;           /* OID of superior type */
        char *at_equality_oid;      /* OID of equality matching rule */
        char *at_ordering_oid;      /* OID of ordering matching rule */
        char *at_substr_oid;        /* OID of substrings matching rule */
        char *at_syntax_oid;        /* OID of syntax of values */
        int  at_syntax_len;         /* Suggested minimum maximum length */
        int  at_single_value;       /* Is single-valued?  */
        int  at_collective;         /* Is collective? */
        int  at_no_user_mod;        /* Are changes forbidden through LDAP? */
        int  at_usage;              /* Usage, see below */
        LDAPSchemaExtensionItem **at_extensions; /* Extensions */
} LDAPAttributeType;

typedef struct ldap_objectclass {
        char *oc_oid;               /* OID */
        char **oc_names;            /* Names */
        char *oc_desc;              /* Description */
        int  oc_obsolete;           /* Is obsolete? */
        char **oc_sup_oids;         /* OIDs of superior classes */
        int  oc_kind;               /* Kind, see below */
        char **oc_at_oids_must;     /* OIDs of required attribute types */
        char **oc_at_oids_may;      /* OIDs of optional attribute types */
        LDAPSchemaExtensionItem **oc_extensions; /* Extensions */
} LDAPObjectClass;
```

Some integer fields (those described with a question mark) have a truth value, for these fields the possible values are:

LDAP_SCHEMA_NO
The answer to the question is no.

LDAP_SCHEMA_YES
The answer to the question is yes.

For attribute types, the following usages are possible:

LDAP_SCHEMA_USER_APPLICATIONS
the attribute type is non-operational.

LDAP_SCHEMA_DIRECTORY_OPERATION
the attribute type is operational and is pertinent to the directory itself, i.e. it has the same value on all servers that master the entry containing this attribute type.

LDAP_SCHEMA_DISTRIBUTED_OPERATION
the attribute type is operational and is pertinent to replication, shadowing or other distributed directory aspect. TBC.

LDAP_SCHEMA_DSA_OPERATION
the attribute type is operational and is pertinent to the directory server itself, i.e. it may have different values for the same entry when retrieved from different servers that master the entry.

Object classes can be of three kinds:

LDAP_SCHEMA_ABSTRACT
the object class is abstract, i.e. there cannot be entries of this class alone.

LDAP_SCHEMA_STRUCTURAL
the object class is structural, i.e. it describes the main role of the entry. On some servers, once the entry is created the set of structural object classes assigned cannot be changed: none of those present can be removed and none other can be added.

LDAP_SCHEMA_AUXILIARY
the object class is auxiliary, i.e. it is intended to go with other, structural, object classes. These can be added or removed at any time if attribute types are added or removed at the same time as needed by the set of object classes resulting from the operation.

Routines **ldap_xxx2name()** return a canonical name for the definition.

Routines **ldap_xxx2str()** return a string representation in the format described by RFC 4512 of the struct passed in the argument. The string is a newly allocated string that must be freed by the caller. These routines may return NULL if no memory can be allocated for the string.

ldap_scherr2str() returns a NUL-terminated string with a text description of the error found. This is a pointer to a static area, so it must not be freed by the caller. The argument *code* comes from one of the parsing routines and can adopt the following values:

LDAP_SCHERR_OUTOFMEM
Out of memory.

LDAP_SCHERR_UNEXPTOKEN
Unexpected token.

LDAP_SCHERR_NOLEFTPAREN
Missing opening parenthesis.

LDAP_SCHERR_NORIGHTPAREN
Missing closing parenthesis.

LDAP_SCHERR_NODIGIT
Expecting digit.

LDAP_SCHERR_BADNAME
Expecting a name.

LDAP_SCHERR_BADDESC
Bad description.

LDAP_SCHERR_BADSUP
Bad superiors.

LDAP_SCHERR_DUPOPT
Duplicate option.

LDAP_SCHERR_EMPTY
Unexpected end of data.

SEE ALSO

ldap(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_search, ldap_search_s, ldap_search_st, ldap_search_ext, ldap_search_ext_s − Perform an LDAP search operation

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

```
#include <sys/types.h>
#include <ldap.h>

int ldap_search_ext(
        LDAP *ld,
        char *base,
        int scope,
        char *filter,
        char *attrs[],
        int attrsonly,
        LDAPControl **serverctrls,
        LDAPControl **clientctrls,
        struct timeval *timeout,
        int sizelimit,
        int *msgidp );

int ldap_search_ext_s(
        LDAP *ld,
        char *base,
        int scope,
        char *filter,
        char *attrs[],
        int attrsonly,
        LDAPControl **serverctrls,
        LDAPControl **clientctrls,
        struct timeval *timeout,
        int sizelimit,
        LDAPMessage **res );
```

DESCRIPTION

These routines are used to perform LDAP search operations. The **ldap_search_ext_s()** routine does the search synchronously (i.e., not returning until the operation completes), providing a pointer to the resulting LDAP messages at the location pointed to by the *res* parameter.

The **ldap_search_ext()** routine is the asynchronous version, initiating the search and returning the message id of the operation it initiated in the integer pointed to by the *msgidp* parameter.

The *base* parameter is the DN of the entry at which to start the search.

The *scope* parameter is the scope of the search and should be one of LDAP_SCOPE_BASE, to search the object itself, LDAP_SCOPE_ONELEVEL, to search the object's immediate children, LDAP_SCOPE_SUBTREE, to search the object and all its descendants, or LDAP_SCOPE_CHILDREN, to search all of the descendants. Note that the latter requires the server support the LDAP Subordinates Search Scope extension.

The *filter* is a string representation of the filter to apply in the search. The string should conform to the format specified in RFC 4515 as extended by RFC 4526. For instance, "(cn=Jane Doe)". Note that use of the extension requires the server to support the LDAP Absolute True/False Filter extension. NULL may be specified to indicate the library should send the filter (objectClass=*).

The *attrs* parameter is a null-terminated array of attribute descriptions to return from matching entries. If NULL is specified, the return of all user attributes is requested. The description "*"

(LDAP_ALL_USER_ATTRIBUTES) may be used to request all user attributes to be returned. The description "+"(LDAP_ALL_OPERATIONAL_ATTRIBUTES) may be used to request all operational attributes to be returned. Note that this requires the server to support the LDAP All Operational Attribute extension. To request no attributes, the description "1.1" (LDAP_NO_ATTRS) should be listed by itself.

The *attrsonly* parameter should be set to a non-zero value if only attribute descriptions are wanted. It should be set to zero (0) if both attributes descriptions and attribute values are wanted.

The *serverctrls* and *clientctrls* parameters may be used to specify server and client controls, respectively.

The **ldap_search_ext_s()** routine is the synchronous version of **ldap_search_ext().**

It also returns a code indicating success or, in the case of failure, indicating the nature of the failure of the operation. See **ldap_error**(3) for details.

NOTES

Note that both read and list functionality are subsumed by these routines, by using a filter like "(objectclass=*)" and a scope of LDAP_SCOPE_BASE (to emulate read) or LDAP_SCOPE_ONELEVEL (to emulate list).

These routines may dynamically allocate memory. The caller is responsible for freeing such memory using supplied deallocation routines. Return values are contained in <ldap.h>.

DEPRECATED INTERFACES

The **ldap_search()** routine is deprecated in favor of the **ldap_search_ext()** routine. The **ldap_search_s()** and **ldap_search_st()** routines are deprecated in favor of the **ldap_search_ext_s()** routine.

SEE ALSO

ldap(3), **ldap_result**(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

NAME

ldap_search, ldap_search_s, ldap_search_st, ldap_search_ext, ldap_search_ext_s − Perform an LDAP search operation

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

```
#include <sys/types.h>
#include <ldap.h>

int ldap_search_ext(
        LDAP *ld,
        char *base,
        int scope,
        char *filter,
        char *attrs[],
        int attrsonly,
        LDAPControl **serverctrls,
        LDAPControl **clientctrls,
        struct timeval *timeout,
        int sizelimit,
        int *msgidp );

int ldap_search_ext_s(
        LDAP *ld,
        char *base,
        int scope,
        char *filter,
        char *attrs[],
        int attrsonly,
        LDAPControl **serverctrls,
        LDAPControl **clientctrls,
        struct timeval *timeout,
        int sizelimit,
        LDAPMessage **res );
```

DESCRIPTION

These routines are used to perform LDAP search operations. The **ldap_search_ext_s()** routine does the search synchronously (i.e., not returning until the operation completes), providing a pointer to the resulting LDAP messages at the location pointed to by the *res* parameter.

The **ldap_search_ext()** routine is the asynchronous version, initiating the search and returning the message id of the operation it initiated in the integer pointed to by the *msgidp* parameter.

The *base* parameter is the DN of the entry at which to start the search.

The *scope* parameter is the scope of the search and should be one of LDAP_SCOPE_BASE, to search the object itself, LDAP_SCOPE_ONELEVEL, to search the object's immediate children, LDAP_SCOPE_SUBTREE, to search the object and all its descendants, or LDAP_SCOPE_CHILDREN, to search all of the descendants. Note that the latter requires the server support the LDAP Subordinates Search Scope extension.

The *filter* is a string representation of the filter to apply in the search. The string should conform to the format specified in RFC 4515 as extended by RFC 4526. For instance, "(cn=Jane Doe)". Note that use of the extension requires the server to support the LDAP Absolute True/False Filter extension. NULL may be specified to indicate the library should send the filter (objectClass=*).

The *attrs* parameter is a null-terminated array of attribute descriptions to return from matching entries. If NULL is specified, the return of all user attributes is requested. The description "*"

(LDAP_ALL_USER_ATTRIBUTES) may be used to request all user attributes to be returned. The description "+"(LDAP_ALL_OPERATIONAL_ATTRIBUTES) may be used to request all operational attributes to be returned. Note that this requires the server to support the LDAP All Operational Attribute extension. To request no attributes, the description "1.1" (LDAP_NO_ATTRS) should be listed by itself.

The *attrsonly* parameter should be set to a non-zero value if only attribute descriptions are wanted. It should be set to zero (0) if both attributes descriptions and attribute values are wanted.

The *serverctrls* and *clientctrls* parameters may be used to specify server and client controls, respectively.

The **ldap_search_ext_s()** routine is the synchronous version of **ldap_search_ext().**

It also returns a code indicating success or, in the case of failure, indicating the nature of the failure of the operation. See **ldap_error**(3) for details.

NOTES

Note that both read and list functionality are subsumed by these routines, by using a filter like "(objectclass=*)" and a scope of LDAP_SCOPE_BASE (to emulate read) or LDAP_SCOPE_ONELEVEL (to emulate list).

These routines may dynamically allocate memory. The caller is responsible for freeing such memory using supplied deallocation routines. Return values are contained in <ldap.h>.

DEPRECATED INTERFACES

The **ldap_search()** routine is deprecated in favor of the **ldap_search_ext()** routine. The **ldap_search_s()** and **ldap_search_st()** routines are deprecated in favor of the **ldap_search_ext_s()** routine.

Deprecated interfaces generally remain in the library. The macro LDAP_DEPRECATED can be defined to a non-zero value (e.g., -DLDAP_DEPRECATED=1) when compiling program designed to use deprecated interfaces. It is recommended that developers writing new programs, or updating old programs, avoid use of deprecated interfaces. Over time, it is expected that documentation (and, eventually, support) for deprecated interfaces to be eliminated.

SEE ALSO

ldap(3), **ldap_result**(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_sort_entries, ldap_sort_values, ldap_sort_strcasecmp − LDAP sorting routines (deprecated)

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

DESCRIPTION

The **ldap_sort_entries**(), **ldap_sort_values**(), and **ldap_sort_strcasecmp**() are deprecated.

SEE ALSO

ldap(3)

ACKNOWLEDGEMENTS

NAME

ldap_sort_entries, ldap_sort_values, ldap_sort_strcasecmp − LDAP sorting routines (deprecated)

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

DESCRIPTION

The **ldap_sort_entries**(), **ldap_sort_values**(), and **ldap_sort_strcasecmp**() are deprecated.

Deprecated interfaces generally remain in the library. The macro LDAP_DEPRECATED can be defined to a non-zero value (e.g., -DLDAP_DEPRECATED=1) when compiling program designed to use deprecated interfaces. It is recommended that developers writing new programs, or updating old programs, avoid use of deprecated interfaces. Over time, it is expected that documentation (and, eventually, support) for deprecated interfaces to be eliminated.

SEE ALSO

ldap(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_sync_init, ldap_sync_init_refresh_only, ldap_sync_init_refresh_and_persist, ldap_sync_poll − LDAP sync routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap_sync.h>

int ldap_sync_init(ldap_sync_t ****ls*, int** *mode***);**

int ldap_sync_init_refresh_only(ldap_sync_t ****ls*);**

int ldap_sync_init_refresh_and_persist(ldap_sync_t ****ls*);**

int ldap_sync_poll(ldap_sync_t ****ls*);**

ldap_sync_t * ldap_sync_initialize(ldap_sync_t ****ls*);**

int ldap_sync_destroy(ldap_sync_t ****ls*, int** *freeit***);**

typedef int (**ldap_sync_search_entry_f***)(ldap_sync_t ****ls***,**
LDAPMessage ****msg*, struct berval ****entryUUID***,**
ldap_sync_refresh_t *phase***);**

typedef int (**ldap_sync_search_reference_f***)(ldap_sync_t ****ls***,**
LDAPMessage ****msg*);**

typedef int (**ldap_sync_intermediate_f***)(ldap_sync_t ****ls***,**
LDAPMessage ****msg*, BerVarray** *syncUUIDs***,**
ldap_sync_refresh_t *phase***);**

typedef int (**ldap_sync_search_result_f***)(ldap_sync_t ****ls***,**
LDAPMessage ****msg*, int** *refreshDeletes***);**

DESCRIPTION

These routines provide an interface to the LDAP Content Synchronization operation (RFC 4533). They require an **ldap_sync_t** structure to be set up with parameters required for various phases of the operation; this includes setting some handlers for special events. All handlers take a pointer to the **ldap_sync_t** structure as the first argument, and a pointer to the **LDAPMessage** structure as received from the server by the client library, plus, occasionally, other specific arguments.

The members of the **ldap_sync_t** structure are:

**char ****ls_base*
The search base; by default, the **BASE** option in **ldap.conf**(5).

int *ls_scope*
The search scope (one of **LDAP_SCOPE_BASE**, **LDAP_SCOPE_ONELEVEL**, **LDAP_SCOPE_SUBORDINATE** or **LDAP_SCOPE_SUBTREE**; see **ldap.h** for details).

**char ****ls_filter*
The filter (RFC 4515); by default, **(objectClass=*)**.

**char *****ls_attrs*
The requested attributes; by default **NULL**, indicating all user attributes.

int *ls_timelimit*
The requested time limit (in seconds); by default **0**, to indicate no limit.

int *ls_sizelimit*
The requested size limit (in entries); by default **0**, to indicate no limit.

int *ls_timeout*

The desired timeout during polling with **ldap_sync_poll**(3). A value of **-1** means that polling is blocking, so **ldap_sync_poll**(3) will not return until a message is received; a value of **0** means that polling returns immediately, no matter if any response is available or not; a positive value represents the timeout the **ldap_sync_poll**(3) function will wait for response before returning, unless a message is received; in that case, **ldap_sync_poll**(3) returns as soon as the message is available.

ldap_sync_search_entry_f *ls_search_entry*

A function that is called whenever an entry is returned. The **msg** argument is the **LDAPMessage** that contains the searchResultEntry; it can be parsed using the regular client API routines, like **ldap_get_dn**(3), **ldap_first_attribute**(3), and so on. The **entryUUID** argument contains the entryUUID of the entry. The **phase** argument indicates the type of operation: one of **LDAP_SYNC_CAPI_PRESENT**, **LDAP_SYNC_CAPI_ADD**, **LDAP_SYNC_CAPI_MODIFY**, **LDAP_SYNC_CAPI_DELETE**; in case of **LDAP_SYNC_CAPI_PRESENT** or **LDAP_SYNC_CAPI_DELETE**, only the DN is contained in the *LDAPMessage*; in case of **LDAP_SYNC_CAPI_MODIFY**, the whole entry is contained in the *LDAPMessage*, and the application is responsible of determining the differences between the new view of the entry provided by the caller and the data already known.

ldap_sync_search_reference_f *ls_search_reference*

A function that is called whenever a search reference is returned. The **msg** argument is the **LDAPMessage** that contains the searchResultReference; it can be parsed using the regular client API routines, like **ldap_parse_reference**(3).

ldap_sync_intermediate_f *ls_intermediate*

A function that is called whenever something relevant occurs during the refresh phase of the search, which is marked by an *intermediateResponse* message type. The **msg** argument is the **LDAPMessage** that contains the intermediate response; it can be parsed using the regular client API routines, like **ldap_parse_intermediate**(3). The **syncUUIDs** argument contains an array of UUIDs of the entries that depends on the value of the **phase** argument. In case of **LDAP_SYNC_CAPI_PRESENTS**, the "present" phase is being entered; this means that the following sequence of results will consist in entries in "present" sync state. In case of **LDAP_SYNC_CAPI_DELETES**, the "deletes" phase is being entered; this means that the following sequence of results will consist in entries in "delete" sync state. In case of **LDAP_SYNC_CAPI_PRESENTS_IDSET**, the message contains a set of UUIDs of entries that are present; it replaces a "presents" phase. In case of **LDAP_SYNC_CAPI_DELETES_IDSET**, the message contains a set of UUIDs of entries that have been deleted; it replaces a "deletes" phase. In case of **LDAP_SYNC_CAPI_DONE,** a "presents" phase with "refreshDone" set to "TRUE" has been returned to indicate that the refresh phase of refreshAndPersist is over, and the client should start polling. Except for the **LDAP_SYNC_CAPI_PRESENTS_IDSET** and **LDAP_SYNC_CAPI_DELETES_IDSET** cases, **syncUUIDs** is NULL.

ldap_sync_search_result_f *ls_search_result*

A function that is called whenever a searchResultDone is returned. In refreshAndPersist this can only occur when the server decides that the search must be interrupted. The **msg** argument is the **LDAPMessage** that contains the response; it can be parsed using the regular client API routines, like **ldap_parse_result**(3). The **refreshDeletes** argument is not relevant in this case; it should always be -1.

**void ***ls_private*

A pointer to private data. The client may register here a pointer to data the handlers above may need.

**LDAP ***ls_ld*

A pointer to a LDAP structure that is used to connect to the server. It is the responsibility of the client to initialize the structure and to provide appropriate authentication and security in place.

GENERAL USE

A **ldap_sync_t** structure is initialized by calling **ldap_sync_initialize(3).** This simply clears out the contents of an already existing **ldap_sync_t** structure, and sets appropriate values for some members. After that, the caller is responsible for setting up the connection (member **ls_ld**), eventually setting up transport security (TLS), for binding and any other initialization. The caller must also fill all the documented search-related fields of the **ldap_sync_t** structure.

At the end of a session, the structure can be cleaned up by calling **ldap_sync_destroy**(3), which takes care of freeing all data assuming it was allocated by **ldap_mem***(3) routines. Otherwise, the caller should take care of destroying and zeroing out the documented search-related fields, and call **ldap_sync_destroy**(3) to free undocumented members set by the API.

REFRESH ONLY

The **refreshOnly** functionality is obtained by periodically calling **ldap_sync_init**(3) with mode set to **LDAP_SYNC_REFRESH_ONLY**, or, which is equivalent, by directly calling **ldap_sync_init_refresh_only**(3). The state of the search, and the consistency of the search parameters, is preserved across calls by passing the **ldap_sync_t** structure as left by the previous call.

REFRESH AND PERSIST

The **refreshAndPersist** functionality is obtained by calling **ldap_sync_init**(3) with mode set to **LDAP_SYNC_REFRESH_AND_PERSIST**, or, which is equivalent, by directly calling **ldap_sync_init_refresh_and_persist**(3) and, after a successful return, by repeatedly polling with **ldap_sync_poll**(3) according to the desired pattern.

A client may insert a call to **ldap_sync_poll**(3) into an external loop to check if any modification was returned; in this case, it might be appropriate to set **ls_timeout** to 0, or to set it to a finite, small value. Otherwise, if the client's main purpose consists in waiting for responses, a timeout of -1 is most suitable, so that the function only returns after some data has been received and handled.

ERRORS

All routines return any LDAP error resulting from a lower-level error in the API calls they are based on, or LDAP_SUCCESS in case of success. **ldap_sync_poll**(3) may return **LDAP_SYNC_REFRESH_REQUIRED** if a full refresh is requested by the server. In this case, it is appropriate to call **ldap_sync_init**(3) again, passing the same **ldap_sync_t** structure as resulted from any previous call.

NOTES

SEE ALSO

ldap(3), **ldap_search_ext**(3), **ldap_result**(3)**; RFC 4533** (http://www.rfc-editor.org),

AUTHOR

Designed and implemented by Pierangelo Masarati, based on RFC 4533 and loosely inspired by syncrepl code in **slapd**(8).

ACKNOWLEDGEMENTS

Initially developed by **SysNet s.n.c.** **OpenLDAP** is developed and maintained by The OpenLDAP Project (http://www.openldap.org/). **OpenLDAP** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_sync_init, ldap_sync_init_refresh_only, ldap_sync_init_refresh_and_persist, ldap_sync_poll − LDAP sync routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap_sync.h>

int ldap_sync_init(ldap_sync_t ****ls*, int** *mode***);**

int ldap_sync_init_refresh_only(ldap_sync_t ****ls*);**

int ldap_sync_init_refresh_and_persist(ldap_sync_t ****ls*);**

int ldap_sync_poll(ldap_sync_t ****ls*);**

ldap_sync_t * ldap_sync_initialize(ldap_sync_t ****ls*);**

int ldap_sync_destroy(ldap_sync_t ****ls*, int** *freeit***);**

typedef int (**ldap_sync_search_entry_f***)(ldap_sync_t ****ls***,**
LDAPMessage ****msg*, struct berval ****entryUUID***,**
ldap_sync_refresh_t *phase***);**

typedef int (**ldap_sync_search_reference_f***)(ldap_sync_t ****ls***,**
LDAPMessage ****msg*);**

typedef int (**ldap_sync_intermediate_f***)(ldap_sync_t ****ls***,**
LDAPMessage ****msg*, BerVarray** *syncUUIDs***,**
ldap_sync_refresh_t *phase***);**

typedef int (**ldap_sync_search_result_f***)(ldap_sync_t ****ls***,**
LDAPMessage ****msg*, int** *refreshDeletes***);**

DESCRIPTION

These routines provide an interface to the LDAP Content Synchronization operation (RFC 4533). They require an **ldap_sync_t** structure to be set up with parameters required for various phases of the operation; this includes setting some handlers for special events. All handlers take a pointer to the **ldap_sync_t** structure as the first argument, and a pointer to the **LDAPMessage** structure as received from the server by the client library, plus, occasionally, other specific arguments.

The members of the **ldap_sync_t** structure are:

**char ****ls_base*
The search base; by default, the **BASE** option in **ldap.conf**(5).

int *ls_scope*
The search scope (one of **LDAP_SCOPE_BASE**, **LDAP_SCOPE_ONELEVEL**, **LDAP_SCOPE_SUBORDINATE** or **LDAP_SCOPE_SUBTREE**; see **ldap.h** for details).

**char ****ls_filter*
The filter (RFC 4515); by default, **(objectClass=*)**.

**char *****ls_attrs*
The requested attributes; by default **NULL**, indicating all user attributes.

int *ls_timelimit*
The requested time limit (in seconds); by default **0**, to indicate no limit.

int *ls_sizelimit*
The requested size limit (in entries); by default **0**, to indicate no limit.

int *ls_timeout*

The desired timeout during polling with **ldap_sync_poll**(3). A value of **-1** means that polling is blocking, so **ldap_sync_poll**(3) will not return until a message is received; a value of **0** means that polling returns immediately, no matter if any response is available or not; a positive value represents the timeout the **ldap_sync_poll**(3) function will wait for response before returning, unless a message is received; in that case, **ldap_sync_poll**(3) returns as soon as the message is available.

ldap_sync_search_entry_f *ls_search_entry*

A function that is called whenever an entry is returned. The **msg** argument is the **LDAPMessage** that contains the searchResultEntry; it can be parsed using the regular client API routines, like **ldap_get_dn**(3), **ldap_first_attribute**(3), and so on. The **entryUUID** argument contains the entryUUID of the entry. The **phase** argument indicates the type of operation: one of **LDAP_SYNC_CAPI_PRESENT**, **LDAP_SYNC_CAPI_ADD**, **LDAP_SYNC_CAPI_MODIFY**, **LDAP_SYNC_CAPI_DELETE**; in case of **LDAP_SYNC_CAPI_PRESENT** or **LDAP_SYNC_CAPI_DELETE**, only the DN is contained in the *LDAPMessage*; in case of **LDAP_SYNC_CAPI_MODIFY**, the whole entry is contained in the *LDAPMessage*, and the application is responsible of determining the differences between the new view of the entry provided by the caller and the data already known.

ldap_sync_search_reference_f *ls_search_reference*

A function that is called whenever a search reference is returned. The **msg** argument is the **LDAPMessage** that contains the searchResultReference; it can be parsed using the regular client API routines, like **ldap_parse_reference**(3).

ldap_sync_intermediate_f *ls_intermediate*

A function that is called whenever something relevant occurs during the refresh phase of the search, which is marked by an *intermediateResponse* message type. The **msg** argument is the **LDAPMessage** that contains the intermediate response; it can be parsed using the regular client API routines, like **ldap_parse_intermediate**(3). The **syncUUIDs** argument contains an array of UUIDs of the entries that depends on the value of the **phase** argument. In case of **LDAP_SYNC_CAPI_PRESENTS**, the "present" phase is being entered; this means that the following sequence of results will consist in entries in "present" sync state. In case of **LDAP_SYNC_CAPI_DELETES**, the "deletes" phase is being entered; this means that the following sequence of results will consist in entries in "delete" sync state. In case of **LDAP_SYNC_CAPI_PRESENTS_IDSET**, the message contains a set of UUIDs of entries that are present; it replaces a "presents" phase. In case of **LDAP_SYNC_CAPI_DELETES_IDSET**, the message contains a set of UUIDs of entries that have been deleted; it replaces a "deletes" phase. In case of **LDAP_SYNC_CAPI_DONE,** a "presents" phase with "refreshDone" set to "TRUE" has been returned to indicate that the refresh phase of refreshAndPersist is over, and the client should start polling. Except for the **LDAP_SYNC_CAPI_PRESENTS_IDSET** and **LDAP_SYNC_CAPI_DELETES_IDSET** cases, **syncUUIDs** is NULL.

ldap_sync_search_result_f *ls_search_result*

A function that is called whenever a searchResultDone is returned. In refreshAndPersist this can only occur when the server decides that the search must be interrupted. The **msg** argument is the **LDAPMessage** that contains the response; it can be parsed using the regular client API routines, like **ldap_parse_result**(3). The **refreshDeletes** argument is not relevant in this case; it should always be -1.

**void ***ls_private*

A pointer to private data. The client may register here a pointer to data the handlers above may need.

**LDAP ***ls_ld*

A pointer to a LDAP structure that is used to connect to the server. It is the responsibility of the client to initialize the structure and to provide appropriate authentication and security in place.

GENERAL USE

A **ldap_sync_t** structure is initialized by calling **ldap_sync_initialize(3).** This simply clears out the contents of an already existing **ldap_sync_t** structure, and sets appropriate values for some members. After that, the caller is responsible for setting up the connection (member **ls_ld**), eventually setting up transport security (TLS), for binding and any other initialization. The caller must also fill all the documented search-related fields of the **ldap_sync_t** structure.

At the end of a session, the structure can be cleaned up by calling **ldap_sync_destroy**(3), which takes care of freeing all data assuming it was allocated by **ldap_mem***(3) routines. Otherwise, the caller should take care of destroying and zeroing out the documented search-related fields, and call **ldap_sync_destroy**(3) to free undocumented members set by the API.

REFRESH ONLY

The **refreshOnly** functionality is obtained by periodically calling **ldap_sync_init**(3) with mode set to **LDAP_SYNC_REFRESH_ONLY**, or, which is equivalent, by directly calling **ldap_sync_init_refresh_only**(3). The state of the search, and the consistency of the search parameters, is preserved across calls by passing the **ldap_sync_t** structure as left by the previous call.

REFRESH AND PERSIST

The **refreshAndPersist** functionality is obtained by calling **ldap_sync_init**(3) with mode set to **LDAP_SYNC_REFRESH_AND_PERSIST**, or, which is equivalent, by directly calling **ldap_sync_init_refresh_and_persist**(3) and, after a successful return, by repeatedly polling with **ldap_sync_poll**(3) according to the desired pattern.

A client may insert a call to **ldap_sync_poll**(3) into an external loop to check if any modification was returned; in this case, it might be appropriate to set **ls_timeout** to 0, or to set it to a finite, small value. Otherwise, if the client's main purpose consists in waiting for responses, a timeout of -1 is most suitable, so that the function only returns after some data has been received and handled.

ERRORS

All routines return any LDAP error resulting from a lower-level error in the API calls they are based on, or LDAP_SUCCESS in case of success. **ldap_sync_poll**(3) may return **LDAP_SYNC_REFRESH_REQUIRED** if a full refresh is requested by the server. In this case, it is appropriate to call **ldap_sync_init**(3) again, passing the same **ldap_sync_t** structure as resulted from any previous call.

NOTES

SEE ALSO

ldap(3), **ldap_search_ext**(3), **ldap_result**(3)**; RFC 4533** (http://www.rfc-editor.org),

AUTHOR

Designed and implemented by Pierangelo Masarati, based on RFC 4533 and loosely inspired by syncrepl code in **slapd**(8).

ACKNOWLEDGEMENTS

Initially developed by **SysNet s.n.c.** **OpenLDAP** is developed and maintained by The OpenLDAP Project (http://www.openldap.org/). **OpenLDAP** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_start_tls, ldap_start_tls_s, ldap_tls_inplace, ldap_install_tls − LDAP TLS initialization routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_start_tls(LDAP ****ld*);**

int ldap_start_tls_s(LDAP ****ld*, LDAPControl ****serverctrls***, LDAPControl ****clientctrls***);**

int ldap_tls_inplace(LDAP ****ld*);**

int ldap_install_tls(LDAP ****ld*);**

DESCRIPTION

These routines are used to initiate TLS processing on an LDAP session. **ldap_start_tls_s**() sends a StartTLS request to a server, waits for the reply, and then installs TLS handlers on the session if the request succeeded. The routine returns **LDAP_SUCCESS** if everything succeeded, otherwise it returns an LDAP error code. **ldap_start_tls**() sends a StartTLS request to a server and does nothing else. It returns **LDAP_SUCCESS** if the request was sent successfully. **ldap_tls_inplace**() returns 1 if TLS handlers have been installed on the specified session, 0 otherwise. **ldap_install_tls**() installs the TLS handlers on the given session. It returns **LDAP_LOCAL_ERROR** if TLS is already installed.

SEE ALSO

ldap(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

NAME

ldap_start_tls, ldap_start_tls_s, ldap_tls_inplace, ldap_install_tls − LDAP TLS initialization routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_start_tls(LDAP ****ld*);**

int ldap_start_tls_s(LDAP ****ld*, LDAPControl ****serverctrls***, LDAPControl ****clientctrls***);**

int ldap_tls_inplace(LDAP ****ld*);**

int ldap_install_tls(LDAP ****ld*);**

DESCRIPTION

These routines are used to initiate TLS processing on an LDAP session. **ldap_start_tls_s**() sends a Start-TLS request to a server, waits for the reply, and then installs TLS handlers on the session if the request succeeded. The routine returns **LDAP_SUCCESS** if everything succeeded, otherwise it returns an LDAP error code. **ldap_start_tls**() sends a StartTLS request to a server and does nothing else. It returns **LDAP_SUCCESS** if the request was sent successfully. **ldap_tls_inplace**() returns 1 if TLS handlers have been installed on the specified session, 0 otherwise. **ldap_install_tls**() installs the TLS handlers on the given session. It returns **LDAP_LOCAL_ERROR** if TLS is already installed.

SEE ALSO

ldap(3), **ldap_error**(3)

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldap_is_ldap_url, ldap_url_parse, ldap_free_urldesc − LDAP Uniform Resource Locator routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_is_ldap_url(const char *url)

int ldap_url_parse(const char *url, LDAPURLDesc **ludpp)

```
typedef struct ldap_url_desc {
   char *      lud_scheme;     /* URI scheme */
   char *      lud_host;       /* LDAP host to contact */
   int         lud_port;       /* port on host */
   char *      lud_dn;         /* base for search */
   char **     lud_attrs;      /* list of attributes */
   int         lud_scope;      /* a LDAP_SCOPE_... value */
   char *      lud_filter;     /* LDAP search filter */
   char **     lud_exts;       /* LDAP extensions */
   int         lud_crit_exts;  /* true if any extension is critical */
   /* may contain additional fields for internal use */
} LDAPURLDesc;
```

void ldap_free_urldesc(LDAPURLDesc *ludp);

DESCRIPTION

These routines support the use of LDAP URLs (Uniform Resource Locators) as detailed in RFC 4516. LDAP URLs look like this:

ldap://*hostport*/*dn*[**?***attrs*[**?***scope*[**?***filter*[**?***exts*]]]]

where:
hostport is a host name with an optional ":portnumber"
dn is the search base
attrs is a comma separated list of attributes to request
scope is one of these three strings:
base one sub (default=base)
filter is filter
exts are recognized set of LDAP and/or API extensions.

Example:
ldap://ldap.example.net/dc=example,dc=net?cn,sn?sub?(cn=*)

URLs that are wrapped in angle-brackets and/or preceded by "URL:" are also tolerated. Alternative LDAP schemes such as ldaps:// and ldapi:// may be parsed using the below routines as well.

ldap_is_ldap_url() returns a non-zero value if *url* looks like an LDAP URL (as opposed to some other kind of URL). It can be used as a quick check for an LDAP URL; the **ldap_url_parse()** routine should be used if a more thorough check is needed.

ldap_url_parse() breaks down an LDAP URL passed in *url* into its component pieces. If successful, zero is returned, an LDAP URL description is allocated, filled in, and *ludpp* is set to point to it. If an error occurs, a non-zero URL error code is returned.

ldap_free_urldesc() should be called to free an LDAP URL description that was obtained from a call to **ldap_url_parse().**

SEE ALSO

ldap(3)
RFC 4516 <http://www.rfc-editor.org/rfc/rfc4516.txt>

ACKNOWLEDGEMENTS

NAME

ldap_is_ldap_url, ldap_url_parse, ldap_free_urldesc − LDAP Uniform Resource Locator routines

LIBRARY

OpenLDAP LDAP (libldap, -lldap)

SYNOPSIS

#include <ldap.h>

int ldap_is_ldap_url(const char *url)

int ldap_url_parse(const char *url, LDAPURLDesc **ludpp)

```
typedef struct ldap_url_desc {
   char *      lud_scheme;     /* URI scheme */
   char *      lud_host;       /* LDAP host to contact */
   int         lud_port;       /* port on host */
   char *      lud_dn;         /* base for search */
   char **     lud_attrs;      /* list of attributes */
   int         lud_scope;      /* a LDAP_SCOPE_... value */
   char *      lud_filter;     /* LDAP search filter */
   char **     lud_exts;       /* LDAP extensions */
   int         lud_crit_exts;  /* true if any extension is critical */
   /* may contain additional fields for internal use */
} LDAPURLDesc;
```

void ldap_free_urldesc(LDAPURLDesc *ludp);

DESCRIPTION

These routines support the use of LDAP URLs (Uniform Resource Locators) as detailed in RFC 4516. LDAP URLs look like this:

ldap://*hostport*/*dn*[**?***attrs*[**?***scope*[**?***filter*[**?***exts*]]]]

where:
hostport is a host name with an optional ":portnumber"
dn is the search base
attrs is a comma separated list of attributes to request
scope is one of these three strings:
base one sub (default=base)
filter is filter
exts are recognized set of LDAP and/or API extensions.

Example:
ldap://ldap.example.net/dc=example,dc=net?cn,sn?sub?(cn=*)

URLs that are wrapped in angle-brackets and/or preceded by "URL:" are also tolerated. Alternative LDAP schemes such as ldaps:// and ldapi:// may be parsed using the below routines as well.

ldap_is_ldap_url() returns a non-zero value if *url* looks like an LDAP URL (as opposed to some other kind of URL). It can be used as a quick check for an LDAP URL; the **ldap_url_parse()** routine should be used if a more thorough check is needed.

ldap_url_parse() breaks down an LDAP URL passed in *url* into its component pieces. If successful, zero is returned, an LDAP URL description is allocated, filled in, and *ludpp* is set to point to it. If an error occurs, a non-zero URL error code is returned.

ldap_free_urldesc() should be called to free an LDAP URL description that was obtained from a call to **ldap_url_parse().**

SEE ALSO

ldap(3)
RFC 4516 <http://www.rfc-editor.org/rfc/rfc4516.txt>

ACKNOWLEDGEMENTS

OpenLDAP Software is developed and maintained by The OpenLDAP Project <http://www.openldap.org/>. **OpenLDAP Software** is derived from University of Michigan LDAP 3.3 Release.

NAME

ldexp — multiply floating-point number by integral power of 2

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
ldexp(double x, int exp);

float
ldexpf(float x, int exp);
```

DESCRIPTION

The **ldexp**() function multiplies a floating-point number by an integral power of 2.

RETURN VALUES

The **ldexp**() function returns the value of *x* times 2 raised to the power *exp*.

If the input *x* is a NaN, infinity, or 0.0, it is returned unchanged.

If the result would cause an overflow, the global variable *errno* is set to ERANGE and infinity is returned, with the same sign as *x*.

If the result would cause underflow to 0.0, the global variable *errno* is set to ERANGE and the value 0.0 is returned.

SEE ALSO

frexp(3), math(3), modf(3)

STANDARDS

The **ldexp**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

ldiv — return quotient and remainder from division

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

ldiv_t
ldiv(long int num, long int denom);
```

DESCRIPTION

The **ldiv**() function computes the value *num/denom* and returns the quotient and remainder in a structure named *ldiv_t* that contains two *long integer* members named *quot* and *rem*.

SEE ALSO

div(3), lldiv(3), math(3), qdiv(3)

STANDARDS

The **ldiv**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

lgamma, **lgammaf**, **lgamma_r**, **lgammaf_r**, **gamma**, **gammaf**, **gamma_r**, **gammaf_r** — log gamma function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

extern int signgam;

double
lgamma(double x);

float
lgammaf(float x);

double
lgamma_r(double x, int *sign);

float
lgammaf_r(float x, int *sign);

double
gamma(double x);

float
gammaf(float x);

double
gamma_r(double x, int *sign);

float
gammaf_r(float x, int *sign);
```

DESCRIPTION

lgamma(*x*) returns $\ln|\Gamma(x)|$ where

$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\,dt$ for $x > 0$ and
$\Gamma(x) = \pi/(\Gamma(1-x)\sin(\pi x))$ for $x < 1$.

The external integer *signgam* returns the sign of $\Gamma(x)$.

lgamma_r() is a reentrant interface that performs identically to **lgamma**(), differing in that the sign of $\Gamma(x)$ is stored in the location pointed to by the *sign* argument and *signgam* is not modified.

IDIOSYNCRASIES

Do not use the expression "`signgam*exp(lgamma(x))`" to compute g := $\Gamma(x)$. Instead use a program like this (in C):

```
lg = lgamma(x); g = signgam*exp(lg);
```

Only after **lgamma**() has returned can signgam be correct.

RETURN VALUES

lgamma() returns appropriate values unless an argument is out of range. Overflow will occur for sufficiently large positive values, and non-positive integers. On the VAX, the reserved operator is returned, and *errno* is

set to ERANGE.

SEE ALSO

math(3)

HISTORY

The **lgamma** function appeared in 4.3BSD.

NAME

lh_stats, lh_node_stats, lh_node_usage_stats, lh_stats_bio, lh_node_stats_bio, lh_node_usage_stats_bio − LHASH statistics

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/lhash.h>

void lh_stats(LHASH *table, FILE *out);
void lh_node_stats(LHASH *table, FILE *out);
void lh_node_usage_stats(LHASH *table, FILE *out);

void lh_stats_bio(LHASH *table, BIO *out);
void lh_node_stats_bio(LHASH *table, BIO *out);
void lh_node_usage_stats_bio(LHASH *table, BIO *out);
```

DESCRIPTION

The **LHASH** structure records statistics about most aspects of accessing the hash table. This is mostly a legacy of Eric Young writing this library for the reasons of implementing what looked like a nice algorithm rather than for a particular software product.

lh_stats() prints out statistics on the size of the hash table, how many entries are in it, and the number and result of calls to the routines in this library.

lh_node_stats() prints the number of entries for each 'bucket' in the hash table.

lh_node_usage_stats() prints out a short summary of the state of the hash table. It prints the 'load' and the 'actual load'. The load is the average number of data items per 'bucket' in the hash table. The 'actual load' is the average number of items per 'bucket', but only for buckets which contain entries. So the 'actual load' is the average number of searches that will need to find an item in the hash table, while the 'load' is the average number that will be done to record a miss.

lh_stats_bio(), *lh_node_stats_bio()* and *lh_node_usage_stats_bio()* are the same as the above, except that the output goes to a **BIO**.

RETURN VALUES

These functions do not return values.

SEE ALSO

openssl_bio (3), *openssl_lhash* (3)

HISTORY

These functions are available in all versions of SSLeay and OpenSSL.

This manpage is derived from the SSLeay documentation.

NAME

libarchive — functions for reading and writing streaming archives

LIBRARY

library “libarchive”

OVERVIEW

The **libarchive** library provides a flexible interface for reading and writing streaming archive files such as tar and cpio. The library is inherently stream-oriented; readers serially iterate through the archive, writers serially add things to the archive. In particular, note that there is no built-in support for random access nor for in-place modification.

When reading an archive, the library automatically detects the format and the compression. The library currently has read support for:

- old-style tar archives,
- most variants of the POSIX “ustar” format,
- the POSIX “pax interchange” format,
- GNU-format tar archives,
- most common cpio archive formats,
- ISO9660 CD images (with or without RockRidge extensions),
- Zip archives.

The library automatically detects archives compressed with gzip(1), bzip2(1), or compress(1) and decompresses them transparently.

When writing an archive, you can specify the compression to be used and the format to use. The library can write

- POSIX-standard “ustar” archives,
- POSIX “pax interchange format” archives,
- POSIX octet-oriented cpio archives,
- two different variants of shar archives.

Pax interchange format is an extension of the tar archive format that eliminates essentially all of the limitations of historic tar formats in a standard fashion that is supported by POSIX-compliant pax(1) implementations on many systems as well as several newer implementations of tar(1). Note that the default write format will suppress the pax extended attributes for most entries; explicitly requesting pax format will enable those attributes for all entries.

The read and write APIs are accessed through the **archive_read_XXX**() functions and the **archive_write_XXX**() functions, respectively, and either can be used independently of the other.

The rest of this manual page provides an overview of the library operation. More detailed information can be found in the individual manual pages for each API or utility function.

READING AN ARCHIVE

To read an archive, you must first obtain an initialized struct archive object from **archive_read_new**(). You can then modify this object for the desired operations with the various **archive_read_set_XXX**() and **archive_read_support_XXX**() functions. In particular, you will need to invoke appropriate **archive_read_support_XXX**() functions to enable the corresponding compression and format support. Note that these latter functions perform two distinct operations: they cause the corresponding support code to be linked into your program, and they enable the corresponding auto-detect code. Unless you have specific constraints, you will generally want to invoke **archive_read_support_compression_all**() and **archive_read_support_format_all**() to enable auto-detect for all formats and compression types currently supported by the library.

Once you have prepared the struct archive object, you call **archive_read_open**() to actually open the archive and prepare it for reading. There are several variants of this function; the most basic expects you to provide pointers to several functions that can provide blocks of bytes from the archive. There are convenience forms that allow you to specify a filename, file descriptor, *FILE* * object, or a block of memory from which to read the archive data. Note that the core library makes no assumptions about the size of the blocks read; callback functions are free to read whatever block size is most appropriate for the medium.

Each archive entry consists of a header followed by a certain amount of data. You can obtain the next header with **archive_read_next_header**(), which returns a pointer to an struct archive_entry structure with information about the current archive element. If the entry is a regular file, then the header will be followed by the file data. You can use **archive_read_data**() (which works much like the read(2) system call) to read this data from the archive. You may prefer to use the higher-level **archive_read_data_skip**(), which reads and discards the data for this entry, **archive_read_data_to_buffer**(), which reads the data into an in-memory buffer, **archive_read_data_to_file**(), which copies the data to the provided file descriptor, or **archive_read_extract**(), which recreates the specified entry on disk and copies data from the archive. In particular, note that **archive_read_extract**() uses the struct archive_entry structure that you provide it, which may differ from the entry just read from the archive. In particular, many applications will want to override the pathname, file permissions, or ownership.

Once you have finished reading data from the archive, you should call **archive_read_close**() to close the archive, then call **archive_read_finish**() to release all resources, including all memory allocated by the library.

The archive_read(3) manual page provides more detailed calling information for this API.

WRITING AN ARCHIVE

You use a similar process to write an archive. The **archive_write_new**() function creates an archive object useful for writing, the various **archive_write_set_XXX**() functions are used to set parameters for writing the archive, and **archive_write_open**() completes the setup and opens the archive for writing.

Individual archive entries are written in a three-step process: You first initialize a struct archive_entry structure with information about the new entry. At a minimum, you should set the pathname of the entry and provide a *struct stat* with a valid *st_mode* field, which specifies the type of object and *st_size* field, which specifies the size of the data portion of the object. The **archive_write_header**() function actually writes the header data to the archive. You can then use **archive_write_data**() to write the actual data.

After all entries have been written, use the **archive_write_finish**() function to release all resources.

The archive_write(3) manual page provides more detailed calling information for this API.

DESCRIPTION

Detailed descriptions of each function are provided by the corresponding manual pages.

All of the functions utilize an opaque struct archive datatype that provides access to the archive contents.

The struct archive_entry structure contains a complete description of a single archive entry. It uses an opaque interface that is fully documented in archive_entry(3).

Users familiar with historic formats should be aware that the newer variants have eliminated most restrictions on the length of textual fields. Clients should not assume that filenames, link names, user names, or group names are limited in length. In particular, pax interchange format can easily accommodate pathnames in arbitrary character sets that exceed *PATH_MAX*.

RETURN VALUES

Most functions return zero on success, non-zero on error. The return value indicates the general severity of the error, ranging from **ARCHIVE_WARN**, which indicates a minor problem that should probably be reported to the user, to **ARCHIVE_FATAL**, which indicates a serious problem that will prevent any further operations on this archive. On error, the **archive_errno**() function can be used to retrieve a numeric error code (see errno(2)). The **archive_error_string**() returns a textual error message suitable for display.

archive_read_new() and **archive_write_new**() return pointers to an allocated and initialized struct archive object.

archive_read_data() and **archive_write_data**() return a count of the number of bytes actually read or written. A value of zero indicates the end of the data for this entry. A negative value indicates an error, in which case the **archive_errno**() and **archive_error_string**() functions can be used to obtain more information.

ENVIRONMENT

There are character set conversions within the archive_entry(3) functions that are impacted by the currently-selected locale.

SEE ALSO

tar(1), archive_entry(3), archive_read(3), archive_util(3), archive_write(3), tar(5)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

BUGS

Some archive formats support information that is not supported by struct archive_entry. Such information cannot be fully archived or restored using this library. This includes, for example, comments, character sets, or the arbitrary key/value pairs that can appear in pax interchange format archives.

Conversely, of course, not all of the information that can be stored in an struct archive_entry is supported by all formats. For example, cpio formats do not support nanosecond timestamps; old tar formats do not support large device numbers.

NAME

libarchive_internals — description of libarchive internal interfaces

OVERVIEW

The **libarchive** library provides a flexible interface for reading and writing streaming archive files such as tar and cpio. Internally, it follows a modular layered design that should make it easy to add new archive and compression formats.

GENERAL ARCHITECTURE

Externally, libarchive exposes most operations through an opaque, object-style interface. The archive_entry(1) objects store information about a single filesystem object. The rest of the library provides facilities to write archive_entry(1) objects to archive files, read them from archive files, and write them to disk. (There are plans to add a facility to read archive_entry(1) objects from disk as well.)

The read and write APIs each have four layers: a public API layer, a format layer that understands the archive file format, a compression layer, and an I/O layer. The I/O layer is completely exposed to clients who can replace it entirely with their own functions.

In order to provide as much consistency as possible for clients, some public functions are virtualized. Eventually, it should be possible for clients to open an archive or disk writer, and then use a single set of code to select and write entries, regardless of the target.

READ ARCHITECTURE

From the outside, clients use the archive_read(3) API to manipulate an **archive** object to read entries and bodies from an archive stream. Internally, the **archive** object is cast to an **archive_read** object, which holds all read-specific data. The API has four layers: The lowest layer is the I/O layer. This layer can be overridden by clients, but most clients use the packaged I/O callbacks provided, for example, by archive_read_open_memory(3), and archive_read_open_fd(3). The compression layer calls the I/O layer to read bytes and decompresses them for the format layer. The format layer unpacks a stream of uncompressed bytes and creates **archive_entry** objects from the incoming data. The API layer tracks overall state (for example, it prevents clients from reading data before reading a header) and invokes the format and compression layer operations through registered function pointers. In particular, the API layer drives the format-detection process: When opening the archive, it reads an initial block of data and offers it to each registered compression handler. The one with the highest bid is initialized with the first block. Similarly, the format handlers are polled to see which handler is the best for each archive. (Prior to 2.4.0, the format bidders were invoked for each entry, but this design hindered error recovery.)

I/O Layer and Client Callbacks

The read API goes to some lengths to be nice to clients. As a result, there are few restrictions on the behavior of the client callbacks.

The client read callback is expected to provide a block of data on each call. A zero-length return does indicate end of file, but otherwise blocks may be as small as one byte or as large as the entire file. In particular, blocks may be of different sizes.

The client skip callback returns the number of bytes actually skipped, which may be much smaller than the skip requested. The only requirement is that the skip not be larger. In particular, clients are allowed to return zero for any skip that they don't want to handle. The skip callback must never be invoked with a negative value.

Keep in mind that not all clients are reading from disk: clients reading from networks may provide different-sized blocks on every request and cannot skip at all; advanced clients may use mmap(2) to read the entire file into memory at once and return the entire file to libarchive as a single block; other clients may begin asynchronous I/O operations for the next block on each request.

Decompresssion Layer

The decompression layer not only handles decompression, it also buffers data so that the format handlers see a much nicer I/O model. The decompression API is a two stage peek/consume model. A read_ahead request specifies a minimum read amount; the decompression layer must provide a pointer to at least that much data. If more data is immediately available, it should return more: the format layer handles bulk data reads by asking for a minimum of one byte and then copying as much data as is available.

A subsequent call to the **consume**() function advances the read pointer. Note that data returned from a **read_ahead**() call is guaranteed to remain in place until the next call to **read_ahead**(). Intervening calls to **consume**() should not cause the data to move.

Skip requests must always be handled exactly. Decompression handlers that cannot seek forward should not register a skip handler; the API layer fills in a generic skip handler that reads and discards data.

A decompression handler has a specific lifecycle:

Registration/Configuration

When the client invokes the public support function, the decompression handler invokes the internal **__archive_read_register_compression**() function to provide bid and initialization functions. This function returns **NULL** on error or else a pointer to a **struct decompressor_t**. This structure contains a *void * config* slot that can be used for storing any customization information.

Bid The bid function is invoked with a pointer and size of a block of data. The decompressor can access its config data through the *decompressor* element of the **archive_read** object. The bid function is otherwise stateless. In particular, it must not perform any I/O operations.

The value returned by the bid function indicates its suitability for handling this data stream. A bid of zero will ensure that this decompressor is never invoked. Return zero if magic number checks fail. Otherwise, your initial implementation should return the number of bits actually checked. For example, if you verify two full bytes and three bits of another byte, bid 19. Note that the initial block may be very short; be careful to only inspect the data you are given. (The current decompressors require two bytes for correct bidding.)

Initialize The winning bidder will have its init function called. This function should initialize the remaining slots of the *struct decompressor_t* object pointed to by the *decompressor* element of the *archive_read* object. In particular, it should allocate any working data it needs in the *data* slot of that structure. The init function is called with the block of data that was used for tasting. At this point, the decompressor is responsible for all I/O requests to the client callbacks. The decompressor is free to read more data as and when necessary.

Satisfy I/O requests

The format handler will invoke the *read_ahead*, *consume*, and *skip* functions as needed.

Finish The finish method is called only once when the archive is closed. It should release anything stored in the *data* and *config* slots of the *decompressor* object. It should not invoke the client close callback.

Format Layer

The read formats have a similar lifecycle to the decompression handlers:

Registration

Allocate your private data and initialize your pointers.

Bid Formats bid by invoking the **read_ahead**() decompression method but not calling the **consume**() method. This allows each bidder to look ahead in the input stream. Bidders should not look further ahead than necessary, as long look aheads put pressure on the decompression layer to buffer lots of data. Most formats only require a few hundred bytes of look ahead; look aheads of a few kilobytes are reasonable. (The ISO9660 reader sometimes looks ahead by 48k, which should be considered an upper limit.)

Read header
: The header read is usually the most complex part of any format. There are a few strategies worth mentioning: For formats such as tar or cpio, reading and parsing the header is straightforward since headers alternate with data. For formats that store all header data at the beginning of the file, the first header read request may have to read all headers into memory and store that data, sorted by the location of the file data. Subsequent header read requests will skip forward to the beginning of the file data and return the corresponding header.

Read Data
: The read data interface supports sparse files; this requires that each call return a block of data specifying the file offset and size. This may require you to carefully track the location so that you can return accurate file offsets for each read. Remember that the decompressor will return as much data as it has. Generally, you will want to request one byte, examine the return value to see how much data is available, and possibly trim that to the amount you can use. You should invoke consume for each block just before you return it.

Skip All Data
: The skip data call should skip over all file data and trailing padding. This is called automatically by the API layer just before each header read. It is also called in response to the client calling the public **`data_skip`**() function.

Cleanup On cleanup, the format should release all of its allocated memory.

API Layer

XXX to do XXX

WRITE ARCHITECTURE

The write API has a similar set of four layers: an API layer, a format layer, a compression layer, and an I/O layer. The registration here is much simpler because only one format and one compression can be registered at a time.

I/O Layer and Client Callbacks

XXX To be written XXX

Compression Layer

XXX To be written XXX

Format Layer

XXX To be written XXX

API Layer

XXX To be written XXX

WRITE_DISK ARCHITECTURE

The write_disk API is intended to look just like the write API to clients. Since it does not handle multiple formats or compression, it is not layered internally.

GENERAL SERVICES

The **`archive_read`**, **`archive_write`**, and **`archive_write_disk`** objects all contain an initial **`archive`** object which provides common support for a set of standard services. (Recall that ANSI/ISO C90 guarantees that you can cast freely between a pointer to a structure and a pointer to the first element of that structure.) The **`archive`** object has a magic value that indicates which API this object is associated with, slots for storing error information, and function pointers for virtualized API functions.

MISCELLANEOUS NOTES

Connecting existing archiving libraries into libarchive is generally quite difficult. In particular, many existing libraries strongly assume that you are reading from a file; they seek forwards and backwards as necessary to locate various pieces of information. In contrast, libarchive never seeks backwards in its input, which sometimes requires very different approaches.

For example, libarchive's ISO9660 support operates very differently from most ISO9660 readers. The libarchive support utilizes a work-queue design that keeps a list of known entries sorted by their location in the input. Whenever libarchive's ISO9660 implementation is asked for the next header, checks this list to find the next item on the disk. Directories are parsed when they are encountered and new items are added to the list. This design relies heavily on the ISO9660 image being optimized so that directories always occur earlier on the disk than the files they describe.

Depending on the specific format, such approaches may not be possible. The ZIP format specification, for example, allows archivers to store key information only at the end of the file. In theory, it is possible to create ZIP archives that cannot be read without seeking. Fortunately, such archives are very rare, and libarchive can read most ZIP archives, though it cannot always extract as much information as a dedicated ZIP program.

SEE ALSO

`archive`(3), `archive_entry`(3), `archive_read`(3), `archive_write`(3), `archive_write_disk`(3)

HISTORY

The **`libarchive`** library first appeared in FreeBSD 5.3.

AUTHORS

The **`libarchive`** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

BUGS

NAME

efence − Electric Fence Malloc Debugger

SYNOPSIS

#include <stdlib.h>

void * malloc (size_t size);

void free (void *ptr);

void * realloc (void *ptr, size_t size);

void * calloc (size_t nelem, size_t elsize);

void * memalign (size_t alignment, size_t size);

void * valloc (size_t size);

extern int EF_ALIGNMENT;

extern int EF_PROTECT_BELOW;

extern int EF_PROTECT_FREE;

DESCRIPTION

Electric Fence helps you detect two common programming bugs: software that overruns the boundaries of a malloc() memory allocation, and software that touches a memory allocation that has been released by free(). Unlike other malloc() debuggers, Electric Fence will detect *read* accesses as well as writes, and it will pinpoint the exact instruction that causes an error. It has been in use at Pixar since 1987, and at many other sites for years.

Electric Fence uses the virtual memory hardware of your computer to place an inaccessible memory page immediately after (or before, at the user's option) each memory allocation. When software reads or writes this inaccessible page, the hardware issues a segmentation fault, stopping the program at the offending instruction. It is then trivial to find the erroneous statement using your favorite debugger. In a similar manner, memory that has been released by free() is made inaccessible, and any code that touches it will get a segmentation fault.

Simply linking your application with libefence.a will allow you to detect most, but not all, malloc buffer overruns and accesses of free memory. If you want to be reasonably sure that you've found *all* bugs of this type, you'll have to read and understand the rest of this man page.

USAGE

Link your program with the library **libefence.a .** Make sure you are *not* linking with **-lmalloc, -lmallocdebug,** or with other malloc-debugger or malloc-enhancer libraries. You can only use one at a time. If your system administrator has installed Electric Fence for public use, you'll be able to use the **-lefence** argument to the linker, otherwise you'll have to put the path-name for **libefence.a** in the linker's command line. Some systems will require special arguments to the linker to assure that you are using the Electric Fence malloc() and not the one from your C library. On AIX systems, you may have to use the flags
-bnso -bnodelcsect -bI:/lib/syscalls.exp
On Sun systems running SunOS 4.X, you'll probably have to use **-Bstatic.**

Run your program *using a debugger.* It's easier to work this way than to create a **core** file and post-mortem debug it. Electric Fence can create *huge* core files, and some operating systems will thus take minutes simply to dump core! Some operating systems will not create usable core files from programs that are linked with Electric Fence. If your program has one of the errors detected by Electric Fence, it will get a segmentation fault (SIGSEGV) at the offending instruction. Use the debugger to locate the erroneous statement, and repair it.

GLOBAL AND ENVIRONMENT VARIABLES

Electric Fence has four configuration switches that can be enabled via the shell environment, or by setting the value of global integer variables using a debugger. These switches change what bugs Electric Fence will detect, so it's important that you know how to use them.

EF_ALIGNMENT

This is an integer that specifies the alignment for any memory allocations that will be returned by malloc(), calloc(), and realloc(). The value is specified in bytes, thus a value of 4 will cause memory to be aligned to 32-bit boundaries unless your system doesn't have a 8-bit characters. EF_ALIGNMENT is set to sizeof(int) by default, since that is generally the word-size of your CPU. If your program requires that allocations be aligned to 64-bit boundaries and you have a 32-bit **int** you'll have to set this value to 8. This is the case when compiling with the **-mips2** flag on MIPS-based systems such as those from SGI. The memory allocation that is returned by Electric Fence malloc() is aligned using the value in EF_ALIGNMENT, and *its size the multiple of that value* that is greater than or equal to the requested size. For this reason, you will sometimes want to set EF_ALIGNMENT to 0 (no alignment), so that you can detect overruns of less than your CPU's word size. Be sure to read the section *WORD-ALIGNMENT AND OVERRUN DETECTION* in this manual page before you try this. To change this value, set EF_ALIGNMENT in the shell environment to an integer value, or assign to the global integer variable EF_ALIGNMENT using a debugger.

EF_PROTECT_BELOW

Electric Fence usually places an inaccessible page immediately after each memory allocation, so that software that runs past the end of the allocation will be detected. Setting EF_PROTECT_BELOW to 1 causes Electric Fence to place the inaccessible page *before* the allocation in the address space, so that under-runs will be detected instead of over-runs. When EF_PROTECT_BELOW is set, the EF_ALIGNMENT parameter is ignored. All allocations will be aligned to virtual-memory-page boundaries, and their size will be the exact size that was requested. To change this value, set EF_PROTECT_BELOW in the shell environment to an integer value, or assign to the global integer variable EF_PROTECT_BELOW using a debugger.

EF_PROTECT_FREE

Electric Fence usually returns free memory to a pool from which it may be re-allocated. If you suspect that a program may be touching free memory, set EF_PROTECT_FREE to 1. This will cause Electric Fence to never re-allocate memory once it has been freed, so that any access to free memory will be detected. Some programs will use tremendous amounts of memory when this parameter is set. To change this value, set EF_PROTECT_FREE in the shell environment to an integer value, or assign to the global integer variable EF_PROTECT_FREE using a debugger.

EF_ALLOW_MALLOC_0

By default, Electric Fence traps calls to malloc() with a size of zero, because they are often the result of a software bug. If EF_ALLOW_MALLOC_0 is non-zero, the software will not trap calls to malloc() with a size of zero. To change this value, set EF_ALLOC_MALLOC_0 in the shell environment to an integer value, or assign to the global integer variable EF_ALLOC_MALLOC_0 using a debugger.

WORD-ALIGNMENT AND OVERRUN DETECTION

There is a conflict between the alignment restrictions that malloc() operates under and the debugging strategy used by Electric Fence. When detecting overruns, Electric Fence malloc() allocates two or more virtual memory pages for each allocation. The last page is made inaccessible in such a way that any read, write, or execute access will cause a segmentation fault. Then, Electric Fence malloc() will return an address such that the first byte after the end of the allocation is on the inaccessible page. Thus, any overrun of the allocation will cause a segmentation fault.

It follows that the address returned by malloc() is the address of the inaccessible page minus the size of the memory allocation. Unfortunately, malloc() is required to return *word-aligned* allocations, since many CPUs can only access a word when its address is aligned. The conflict happens when software makes a memory allocation using a size that is not a multiple of the word size, and expects to do word accesses to that allocation. The location of the inaccessible page is fixed by hardware at a word-aligned address. If Electric Fence malloc() is to return an aligned address, it must increase the size of the allocation to a multiple of the word size. In addition, the functions memalign() and valloc() must honor explicit specifications on the alignment of the memory allocation, and this, as well can only be implemented by increasing the

size of the allocation. Thus, there will be situations in which the end of a memory allocation contains some padding space, and accesses of that padding space will not be detected, even if they are overruns.

Electric Fence provides the variable EF_ALIGNMENT so that the user can control the default alignment used by malloc(), calloc(), and realloc(). To debug overruns as small as a single byte, you can set EF_ALIGNMENT to zero. This will result in Electric Fence malloc() returning unaligned addresses for allocations with sizes that are not a multiple of the word size. This is not a problem in most cases, because compilers must pad the size of objects so that alignment restrictions are honored when storing those objects in arrays. The problem surfaces when software allocates odd-sized buffers for objects that must be word-aligned. One case of this is software that allocates a buffer to contain a structure and a string, and the string has an odd size (this example was in a popular TIFF library). If word references are made to un-aligned buffers, you will see a bus error (SIGBUS) instead of a segmentation fault. The only way to fix this is to re-write the offending code to make byte references or not make odd-sized allocations, or to set EF_ALIGNMENT to the word size.

Another example of software incompatible with EF_ALIGNMENT < word-size is the strcmp() function and other string functions on SunOS (and probably Solaris), which make word-sized accesses to character strings, and may attempt to access up to three bytes beyond the end of a string. These result in a segmentation fault (SIGSEGV). The only way around this is to use versions of the string functions that perform byte references instead of word references.

INSTRUCTIONS FOR DEBUGGING YOUR PROGRAM

1. Link with libefence.a as explained above.
2. Run your program in a debugger and fix any overruns or accesses to free memory.
3. Quit the debugger.
4. Set EF_PROTECT_BELOW = 1 in the shell environment.
5. Repeat step 2, this time repairing underruns if they occur.
6. Quit the debugger.
7. Read the restrictions in the section on *WORD-ALIGNMENT AND OVERRUN DETECTION*. See if you can set EF_ALIGNMENT to 0 and repeat step 2. Sometimes this will be too much work, or there will be problems with library routines for which you don't have the source, that will prevent you from doing this.

MEMORY USAGE AND EXECUTION SPEED

Since Electric Fence uses at least two virtual memory pages for each of its allocations, it's a terrible memory hog. I've sometimes found it necessary to add a swap file using swapon(8) so that the system would have enough virtual memory to debug my program. Also, the way we manipulate memory results in various cache and translation buffer entries being flushed with each call to malloc or free. The end result is that your program will be much slower and use more resources while you are debugging it with Electric Fence.

Don't leave libefence.a linked into production software! Use it only for debugging.

PORTING

Electric Fence is written for ANSI C. You should be able to port it with simple changes to the Makefile and to page.c, which contains the memory management primitives . Many POSIX platforms will require only a re-compile. The operating system facilities required to port Electric Fence are:

A way to allocate memory pages
A way to make selected pages inaccessible.
A way to make the pages accessible again.
A way to detect when a program touches an inaccessible page.
A way to print messages.

Please e-mail me a copy of any changes you have to make, so that I can merge them into the distribution.

AUTHOR

Bruce Perens

WARNINGS

I have tried to do as good a job as I can on this software, but I doubt that it is even theoretically possible to make it bug-free. This software has no warranty. It will not detect some bugs that you might expect it to detect, and will indicate that some non-bugs are bugs. Bruce Perens and/or Pixar will not be liable to any claims resulting from the use of this software or the ideas within it. The entire responsibility for its use must be assumed by the user. If you use it and it results in loss of life and/or property, tough. If it leads you on a wild goose chase and you waste two weeks debugging something, too bad. If you can't deal with the above, please don't use the software! I've written this in an attempt to help other people, not to get myself sued or prosecuted.

LICENSE

CONTACTING THE AUTHOR

Bruce Perens
c/o Pixar
1001 West Cutting Blvd., Suite 200
Richmond, CA 94804

Telephone: 510-215-3502
Fax: 510-236-0388
Internet: Bruce@Pixar.com

FILES

/dev/zero: Source of memory pages (via mmap(2)).

SEE ALSO

malloc(3), mmap(2), mprotect(2), swapon(8)

DIAGNOSTICS

Segmentation Fault: Examine the offending statement for violation of the boundaries of a memory allocation.
Bus Error: See the section on *WORD-ALIGNMENT AND OVERRUN DETECTION*. in this manual page.

BUGS

My explanation of the alignment issue could be improved.

Some Sun systems running SunOS 4.1 are reported to signal an access to a protected page with **SIGBUS** rather than **SIGSEGV,** I suspect this is an undocumented feature of a particular Sun hardware version, not just the operating system. On these systems, eftest will fail with a bus error until you modify the Makefile to define **PAGE_PROTECTION_VIOLATED_SIGNAL** as **SIGBUS.**

There are, without doubt, other bugs and porting issues. Please contact me via e-mail if you have any bug reports, ideas, etc.

WHAT'S BETTER

PURIFY, from Purify Systems, does a much better job than Electric Fence, and does much more. It's available at this writing on SPARC and HP. I'm not affiliated with Purify, I just think it's a wonderful product and you should check it out.

NAME

magic_open, **magic_close**, **magic_error**, **magic_file**, **magic_buffer**, **magic_setflags**, **magic_check**, **magic_compile**, **magic_load** — Magic number recognition library.

LIBRARY

Magic Number Recognition Library (libmagic, −lmagic)

SYNOPSIS

```
#include <magic.h>

magic_t
magic_open(int flags);

void
magic_close(magic_t cookie);

const char *
magic_error(magic_t cookie);

int
magic_errno(magic_t cookie);

const char *
magic_file(magic_t cookie, const char *filename);

const char *
magic_buffer(magic_t cookie, const void *buffer, size_t length);

int
magic_setflags(magic_t cookie, int flags);

int
magic_check(magic_t cookie, const char *filename);

int
magic_compile(magic_t cookie, const char *filename);

int
magic_load(magic_t cookie, const char *filename);
```

DESCRIPTION

These functions operate on the magic database file which is described in magic(5).

The function **magic_open**() creates a magic cookie pointer and returns it. It returns NULL if there was an error allocating the magic cookie. The *flags* argument specifies how the other magic functions should behave:

MAGIC_NONE	No special handling.
MAGIC_DEBUG	Print debugging messages to stderr.
MAGIC_SYMLINK	If the file queried is a symlink, follow it.
MAGIC_COMPRESS	If the file is compressed, unpack it and look at the contents.
MAGIC_DEVICES	If the file is a block or character special device, then open the device and try to look in its contents.

`MAGIC_MIME` Return a mime string, instead of a textual description.

`MAGIC_CONTINUE` Return all matches, not just the first.

`MAGIC_CHECK` Check the magic database for consistency and print warnings to stderr.

`MAGIC_PRESERVE_ATIME`
On systems that support `utime`(2) or `utimes`(2), attempt to preserve the access time of files analyzed.

`MAGIC_RAW` Don't translate unprintable characters to a \ooo octal representation.

`MAGIC_ERROR` Treat operating system errors while trying to open files and follow symlinks as real errors, instead of printing them in the magic buffer.

`MAGIC_NO_CHECK_APPTYPE`
Check for `EMX` application type (only on EMX).

`MAGIC_NO_CHECK_ASCII`
Check for various types of ascii files.

`MAGIC_NO_CHECK_COMPRESS`
Don't look for, or inside compressed files.

`MAGIC_NO_CHECK_ELF`
Don't print elf details.

`MAGIC_NO_CHECK_FORTRAN`
Don't look for fortran sequences inside ascii files.

`MAGIC_NO_CHECK_SOFT`
Don't consult magic files.

`MAGIC_NO_CHECK_TAR`
Don't examine tar files.

`MAGIC_NO_CHECK_TOKENS`
Don't look for known tokens inside ascii files.

`MAGIC_NO_CHECK_TROFF`
Don't look for troff sequences inside ascii files.

The **`magic_close`**() function closes the `magic`(5) database and deallocates any resources used.

The **`magic_error`**() function returns a textual explanation of the last error, or NULL if there was no error.

The **`magic_errno`**() function returns the last operating system error number (`errno`(2)) that was encountered by a system call.

The **`magic_file`**() function returns a textual description of the contents of the *`filename`* argument, or NULL if an error occurred. If the *`filename`* is NULL, then stdin is used.

The **`magic_buffer`**() function returns a textual description of the contents of the *`buffer`* argument with *`length`* bytes size.

The **`magic_setflags`**() function, sets the *`flags`* described above.

The **`magic_check`**() function can be used to check the validity of entries in the colon separated database files passed in as *`filename`*, or NULL for the default database. It returns 0 on success and -1 on failure.

The **`magic_compile`**() function can be used to compile the the colon separated list of database files passed in as *`filename`*, or NULL for the default database. It returns 0 on success and -1 on failure. The compiled files created are named from the `basename`(1) of each file argument with ".mgc" appended to it.

The **magic_load**() function must be used to load the the colon separated list of database files passed in as *filename*, or NULL for the default database file before any magic queries can performed.

The default database file is named by the MAGIC environment variable. If that variable is not set, the default database file name is /usr/share/misc/magic.

magic_load() adds ".mime" and/or ".mgc" to the database filename as appropriate.

RETURN VALUES

The function **magic_open**() returns a magic cookie on success and NULL on failure setting errno to an appropriate value. It will set errno to EINVAL if an unsupported value for flags was given. The **magic_load**(), **magic_compile**(), and **magic_check**() functions return 0 on success and -1 on failure. The **magic_file**(), and **magic_buffer**() functions return a string on success and NULL on failure. The **magic_error**() function returns a textual description of the errors of the above functions, or NULL if there was no error. Finally, **magic_setflags**() returns -1 on systems that don't support utime(2), or utimes(2) when MAGIC_PRESERVE_ATIME is set.

FILES

/usr/share/misc/magic.mime	The non-compiled default magic mime database.
/usr/share/misc/magic.mime.mgc	The compiled default magic mime database.
/usr/share/misc/magic	The non-compiled default magic database.
/usr/share/misc/magic.mgc	The compiled default magic database.

SEE ALSO

file(1), magic(5)

AUTHORS

Måns Rullgård Initial libmagic implementation, and configuration. Christos Zoulas API cleanup, error code and allocation handling.

NAME

libradius — RADIUS client library

SYNOPSIS

```
#include <radlib.h>

struct rad_handle *
rad_acct_open(void);

int
rad_add_server(struct rad_handle *h, const char *host, int port,
    const char *secret, int timeout, int max_tries);

struct rad_handle *
rad_auth_open(void);

void
rad_close(struct rad_handle *h);

int
rad_config(struct rad_handle *h, const char *file);

int
rad_continue_send_request(struct rad_handle *h, int selected, int *fd,
    struct timeval *tv);

int
rad_create_request(struct rad_handle *h, int code);

struct in_addr
rad_cvt_addr(const void *data);

uint32_t
rad_cvt_int(const void *data);

char *
rad_cvt_string(const void *data, size_t len);

int
rad_get_attr(struct rad_handle *h, const void **data, size_t *len);

int
rad_get_vendor_attr(uint32_t *vendor, const void **data, size_t *len);

int
rad_init_send_request(struct rad_handle *h, int *fd, struct timeval *tv);

int
rad_put_addr(struct rad_handle *h, int type, struct in_addr addr);

int
rad_put_attr(struct rad_handle *h, int type, const void *data, size_t len);

int
rad_put_int(struct rad_handle *h, int type, uint32_t value);

int
rad_put_string(struct rad_handle *h, int type, const char *str);
```

int
rad_put_message_authentic(*struct rad_handle *h*);

int
rad_put_vendor_addr(*struct rad_handle *h*, *int vendor*, *int type*,
struct in_addr addr);

int
rad_put_vendor_attr(*struct rad_handle *h*, *int vendor*, *int type*,
*const void *data*, *size_t len*);

int
rad_put_vendor_int(*struct rad_handle *h*, *int vendor*, *int type*,
uint32_t value);

int
rad_put_vendor_string(*struct rad_handle *h*, *int vendor*, *int type*,
*const char *str*);

ssize_t
rad_request_authenticator(*struct rad_handle *h*, *char *buf*, *size_t len*);

int
rad_send_request(*struct rad_handle *h*);

*const char **
rad_server_secret(*struct rad_handle *h*);

*u_char **
rad_demangle(*struct rad_handle *h*, *const void *mangled*, *size_t mlen*);

*u_char **
rad_demangle_mppe_key(*struct rad_handle *h*, *const void *mangled*,
size_t mlen, *size_t *len*);

*const char **
rad_strerror(*struct rad_handle *h*);

DESCRIPTION

The **libradius** library implements the client side of the Remote Authentication Dial In User Service (RADIUS). RADIUS, defined in RFCs 2865 and 2866, allows clients to perform authentication and accounting by means of network requests to remote servers.

Initialization

To use the library, an application must first call **rad_auth_open**() or **rad_acct_open**() to obtain a *struct rad_handle **, which provides the context for subsequent operations. The former function is used for RADIUS authentication and the latter is used for RADIUS accounting. Calls to **rad_auth_open**() and **rad_acct_open**() always succeed unless insufficient virtual memory is available. If the necessary memory cannot be allocated, the functions return NULL. For compatibility with earlier versions of this library, **rad_open**() is provided as a synonym for **rad_auth_open**().

Before issuing any RADIUS requests, the library must be made aware of the servers it can contact. The easiest way to configure the library is to call **rad_config**(). **rad_config**() causes the library to read a configuration file whose format is described in radius.conf(5). The pathname of the configuration file is passed as the *file* argument to **rad_config**(). This argument may also be given as NULL, in which case the standard configuration file /etc/radius.conf is used. **rad_config**() returns 0 on success, or −1 if an error occurs.

The library can also be configured programmatically by calls to **rad_add_server**(). The *host* parameter specifies the server host, either as a fully qualified domain name or as a dotted-quad IP address in text form. The *port* parameter specifies the UDP port to contact on the server. If *port* is given as 0, the library looks up the radius/udp or radacct/udp service in the network services(5) database, and uses the port found there. If no entry is found, the library uses the standard RADIUS ports, 1812 for authentication and 1813 for accounting. The shared secret for the server host is passed to the *secret* parameter. It may be any NUL-terminated string of bytes. The RADIUS protocol ignores all but the leading 128 bytes of the shared secret. The timeout for receiving replies from the server is passed to the *timeout* parameter, in units of seconds. The maximum number of repeated requests to make before giving up is passed into the *max_tries* parameter. **rad_add_server**() returns 0 on success, or −1 if an error occurs.

rad_add_server() may be called multiple times, and it may be used together with **rad_config**(). At most 10 servers may be specified. When multiple servers are given, they are tried in round-robin fashion until a valid response is received, or until each server's *max_tries* limit has been reached.

Creating a RADIUS Request

A RADIUS request consists of a code specifying the kind of request, and zero or more attributes which provide additional information. To begin constructing a new request, call **rad_create_request**(). In addition to the usual *struct rad_handle* *, this function takes a *code* parameter which specifies the type of the request. Most often this will be RAD_ACCESS_REQUEST. **rad_create_request**() returns 0 on success, or −1 on if an error occurs.

After the request has been created with **rad_create_request**(), attributes can be attached to it. This is done through calls to **rad_put_addr**(), **rad_put_int**(), and **rad_put_string**(). Each accepts a *type* parameter identifying the attribute, and a value which may be an Internet address, an integer, or a NUL-terminated string, respectively. Alternatively, **rad_put_vendor_addr**(), **rad_put_vendor_int**() or **rad_put_vendor_string**() may be used to specify vendor specific attributes. Vendor specific definitions may be found in <radlib_vs.h>

The library also provides a function **rad_put_attr**() which can be used to supply a raw, uninterpreted attribute. The *data* argument points to an array of bytes, and the *len* argument specifies its length.

It is possible adding the Message-Authenticator to the request. This is an HMAC-MD5 hash of the entire Access-Request packet (see RFC 3579). This attribute must be present in any packet that includes an EAP-Message attribute. It can be added by using the **rad_put_message_authentic**() function. The **libradius** library calculates the HMAC-MD5 hash implicitly before sending the request. If the Message-Authenticator was found inside the response packet, then the packet is silently dropped, if the validation failed. In order to get this feature, the library should be compiled with OpenSSL support.

The **rad_put_X**() functions return 0 on success, or −1 if an error occurs.

Sending the Request and Receiving the Response

After the RADIUS request has been constructed, it is sent either by means of **rad_send_request**() or by a combination of calls to **rad_init_send_request**() and **rad_continue_send_request**().

The **rad_send_request**() function sends the request and waits for a valid reply, retrying the defined servers in round-robin fashion as necessary. If a valid response is received, **rad_send_request**() returns the RADIUS code which specifies the type of the response. This will typically be RAD_ACCESS_ACCEPT, RAD_ACCESS_REJECT, or RAD_ACCESS_CHALLENGE. If no valid response is received, **rad_send_request**() returns −1.

As an alternative, if you do not wish to block waiting for a response, **rad_init_send_request**() and **rad_continue_send_request**() may be used instead. If a reply is received from the RADIUS server or a timeout occurs, these functions return a value as described for **rad_send_request**(). Otherwise, a value of zero is returned and the values pointed to by *fd* and *tv* are set to the descriptor and timeout that

should be passed to `select`(2).

rad_init_send_request() must be called first, followed by repeated calls to **rad_continue_send_request**() as long as a return value of zero is given. Between each call, the application should call `select`(2), passing **fd* as a read descriptor and timing out after the interval specified by *tv*. When `select`(2) returns, **rad_continue_send_request**() should be called with *selected* set to a non-zero value if `select`(2) indicated that the descriptor is readable.

Like RADIUS requests, each response may contain zero or more attributes. After a response has been received successfully by **rad_send_request**() or **rad_continue_send_request**(), its attributes can be extracted one by one using **rad_get_attr**(). Each time **rad_get_attr**() is called, it gets the next attribute from the current response, and stores a pointer to the data and the length of the data via the reference parameters *data* and *len*, respectively. Note that the data resides in the response itself, and must not be modified. A successful call to **rad_get_attr**() returns the RADIUS attribute type. If no more attributes remain in the current response, **rad_get_attr**() returns 0. If an error such as a malformed attribute is detected, −1 is returned.

If **rad_get_attr**() returns `RAD_VENDOR_SPECIFIC`, **rad_get_vendor_attr**() may be called to determine the vendor. The vendor specific RADIUS attribute type is returned. The reference parameters *data* and *len* (as returned from **rad_get_attr**()) are passed to **rad_get_vendor_attr**(), and are adjusted to point to the vendor specific attribute data.

The common types of attributes can be decoded using **rad_cvt_addr**(), **rad_cvt_int**(), and **rad_cvt_string**(). These functions accept a pointer to the attribute data, which should have been obtained using **rad_get_attr**() and optionally **rad_get_vendor_attr**(). In the case of **rad_cvt_string**(), the length *len* must also be given. These functions interpret the attribute as an Internet address, an integer, or a string, respectively, and return its value. **rad_cvt_string**() returns its value as a `NUL`-terminated string in dynamically allocated memory. The application should free the string using `free`(3) when it is no longer needed.

If insufficient virtual memory is available, **rad_cvt_string**() returns `NULL`. **rad_cvt_addr**() and **rad_cvt_int**() cannot fail.

The **rad_request_authenticator**() function may be used to obtain the Request-Authenticator attribute value associated with the current RADIUS server according to the supplied rad_handle. The target buffer *buf* of length *len* must be supplied and should be at least 16 bytes. The return value is the number of bytes written to *buf* or −1 to indicate that *len* was not large enough.

The **rad_server_secret**() returns the secret shared with the current RADIUS server according to the supplied rad_handle.

The **rad_demangle**() function demangles attributes containing passwords and MS-CHAPv1 MPPE-Keys. The return value is `NULL` on failure, or the plaintext attribute. This value should be freed using `free`(3) when it is no longer needed.

The **rad_demangle_mppe_key**() function demangles the send- and recv-keys when using MPPE (see RFC 2548). The return value is `NULL` on failure, or the plaintext attribute. This value should be freed using `free`(3) when it is no longer needed.

Obtaining Error Messages

Those functions which accept a *struct rad_handle ** argument record an error message if they fail. The error message can be retrieved by calling **rad_strerror**(). The message text is overwritten on each new error for the given *struct rad_handle **. Thus the message must be copied if it is to be preserved through subsequent library calls using the same handle.

Cleanup
To free the resources used by the RADIUS library, call **rad_close**().

RETURN VALUES

The following functions return a non-negative value on success. If they detect an error, they return −1 and record an error message which can be retrieved using **rad_strerror**().

rad_add_server()
rad_config()
rad_create_request()
rad_get_attr()
rad_put_addr()
rad_put_attr()
rad_put_int()
rad_put_string()
rad_put_message_authentic()
rad_init_send_request()
rad_continue_send_request()
rad_send_request()

The following functions return a non-NULL pointer on success. If they are unable to allocate sufficient virtual memory, they return NULL, without recording an error message.

rad_acct_open()
rad_auth_open()
rad_cvt_string()

The following functions return a non-NULL pointer on success. If they fail, they return NULL, with recording an error message.

rad_demangle()
rad_demangle_mppe_key()

FILES

/etc/radius.conf

SEE ALSO

radius.conf(5)

C. Rigney, et al, *Remote Authentication Dial In User Service (RADIUS)*, RFC 2865.

C. Rigney, *RADIUS Accounting*, RFC 2866.

G. Zorn, *Microsoft Vendor-specific RADIUS attributes*, RFC 2548.

C. Rigney, et al, *RADIUS extensions*, RFC 2869.

AUTHORS

This software was originally written by John Polstra, and donated to the FreeBSD project by Juniper Networks, Inc. Oleg Semyonov subsequently added the ability to perform RADIUS accounting. Later additions and changes by Michael Bretterklieber.

NAME

link_addr, **link_ntoa** — elementary address specification routines for link level access

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if_dl.h>

void
link_addr(const char *addr, struct sockaddr_dl *sdl);

char *
link_ntoa(const struct sockaddr_dl *sdl);
```

DESCRIPTION

The routine **link_addr**() interprets character strings representing link-level addresses, returning binary information suitable for use in system calls. The routine **link_ntoa**() takes a link-level address and returns an ASCII string representing some of the information present, including the link level address itself, and the interface name or number, if present. This facility is experimental and is still subject to change.

Prior to a call to **link_addr**(), *sdl->sdl_len* must be initialized to the size of the link-level socket structure, typically *sizeof(struct sockaddr_dl)*.

For **link_addr**(), the string *addr* may contain an optional network interface identifier of the form "name unit-number", suitable for the first argument to `ifconfig`(8), followed in all cases by a colon and an interface address in the form of groups of hexadecimal digits separated by periods. Each group represents a byte of address; address bytes are filled left to right from low order bytes through high order bytes.

Thus `le0:8.0.9.13.d.30` represents an ethernet address to be transmitted on the first Lance ethernet interface.

RETURN VALUES

link_ntoa() always returns a null terminated string. **link_addr**() has no return value (See **BUGS**).

SEE ALSO

`ethers`(3), `iso`(4)

HISTORY

The **link_addr**() and **link_ntoa**() functions appeared in 4.3BSD−Reno.

BUGS

The returned values for **link_ntoa**() reside in a static memory area.

The function **link_addr**() should diagnose improperly formed input, and there should be an unambiguous way to recognize this.

If the *sdl_len* field of the link socket address *sdl* is 0, **link_ntoa**() will not insert a colon before the interface address bytes. If this translated address is given to **link_addr**() without inserting an initial colon, the latter will not interpret it correctly.

NAME

lio_listio — list directed I/O (REALTIME)

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <aio.h>

int
lio_listio(int mode, struct aiocb * const list[], int nent,
    struct sigevent *sig);
```

DESCRIPTION

The **lio_listio**() function initiates a list of I/O requests with a single function call. The *list* argument is an array of pointers to *aiocb* structures describing each operation to perform, with *nent* elements. NULL elements are ignored.

The *aio_lio_opcode* field of each *aiocb* specifies the operation to be performed. The following operations are supported:

LIO_READ Read data as if by a call to aio_read(3).

LIO_NOP No operation.

LIO_WRITE Write data as if by a call to aio_write(3).

If the *mode* argument is LIO_WAIT, **lio_listio**() does not return until all the requested operations have been completed. If *mode* is LIO_NOWAIT, the requests are processed asynchronously, and the signal specified by *sig* is sent when all operations have completed. If *sig* is NULL, the calling process is not notified of I/O completion.

The order in which the requests are carried out is not specified, and there is no guarantee that they will be executed sequentially.

RETURN VALUES

If *mode* is LIO_WAIT, the **lio_listio**() function returns 0 if the operations completed successfully, otherwise −1.

If *mode* is LIO_NOWAIT, the **lio_listio**() function returns 0 if the operations are successfully queued, otherwise −1.

ERRORS

The **lio_listio**() function will fail if:

[EAGAIN] There are not enough resources to enqueue the requests.

[EAGAIN] The request would cause the system-wide limit AIO_MAX to be exceeded.

[EINTR] A signal interrupted the system call before it could be completed.

[EINVAL] The *mode* argument is neither LIO_WAIT nor LIO_NOWAIT, or *nent* is greater than AIO_LISTIO_MAX.

[EIO] One or more requests failed.

In addition, the **lio_listio**() function may fail for any of the reasons listed for aio_read(3) and aio_write(3).

If **lio_listio**() succeeds, or fails with an error code of EAGAIN, EINTR, or EIO, some of the requests may have been initiated. The caller should check the error status of each *aiocb* structure individually by calling aio_error(3).

SEE ALSO

read(2), siginfo(2), write(2), aio_error(3), aio_read(3), aio_write(3)

STANDARDS

The **lio_listio**() function is expected to conform to IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

llabs — return the absolute value of a long long integer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

long long int
llabs(long long int j);
```

DESCRIPTION

The **llabs**() function returns the absolute value of the long long integer *j*.

SEE ALSO

abs(3), cabs(3), floor(3), labs(3), math(3)

STANDARDS

The **llabs**() function conforms to ISO/IEC 9899:1999 (“ISO C99”).

BUGS

The absolute value of the most negative integer remains negative.

NAME

lldiv — return quotient and remainder from division

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

lldiv_t
lldiv(long long int num, long long int denom);
```

DESCRIPTION

The **lldiv**() function computes the value *num/denom* and returns the quotient and remainder in a structure named *lldiv_t* that contains two *long long integer* members named *quot* and *rem*.

SEE ALSO

div(3), ldiv(3), math(3), qdiv(3)

STANDARDS

The **lldiv**() function conforms to ISO/IEC 9899:1999 (“ISO C99”).

NAME

lockf — record locking on files

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
lockf(int filedes, int function, off_t size);
```

DESCRIPTION

The **lockf**() function allows sections of a file to be locked with advisory-mode locks. Calls to **lockf**() from other processes which attempt to lock the locked file section will either return an error value or block until the section becomes unlocked. All the locks for a process are removed when the process terminates.

The argument *filedes* is an open file descriptor. The file descriptor must have been opened either for write-only (O_WRONLY) or read/write (O_RDWR) operation.

The *function* argument is a control value which specifies the action to be taken. The permissible values for *function* are as follows:

Function	Description
F_ULOCK	unlock locked sections
F_LOCK	lock a section for exclusive use
F_TLOCK	test and lock a section for exclusive use
F_TEST	test a section for locks by other processes

F_ULOCK removes locks from a section of the file; F_LOCK and F_TLOCK both lock a section of a file if the section is available; F_TEST detects if a lock by another process is present on the specified section.

The *size* argument is the number of contiguous bytes to be locked or unlocked. The section to be locked or unlocked starts at the current offset in the file and extends forward for a positive size or backward for a negative size (the preceding bytes up to but not including the current offset). However, it is not permitted to lock a section that starts or extends before the beginning of the file. If *size* is 0, the section from the current offset through the largest possible file offset is locked (that is, from the current offset through the present or any future end-of-file).

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent locked sections would occur, the sections are combined into a single locked section. If the request would cause the number of locks to exceed a system-imposed limit, the request will fail.

F_LOCK and F_TLOCK requests differ only by the action taken if the section is not available. F_LOCK blocks the calling process until the section is available. F_TLOCK makes the function fail if the section is already locked by another process.

File locks are released on first close by the locking process of any file descriptor for the file.

F_ULOCK requests release (wholly or in part) one or more locked sections controlled by the process. Locked sections will be unlocked starting at the current file offset through *size* bytes or to the end of file if size is 0. When all of a locked section is not released (that is, when the beginning or end of the area to be unlocked falls within a locked section), the remaining portions of that section are still locked by the process. Releasing the center portion of a locked section will cause the remaining locked beginning and end portions to become two separate locked sections. If the request would cause the number of locks in the system to exceed a system-imposed limit, the request will fail.

An F_ULOCK request in which size is non-zero and the offset of the last byte of the requested section is the maximum value for an object of type off_t, when the process has an existing lock in which size is 0 and which includes the last byte of the requested section, will be treated as a request to unlock from the start of the requested section with a size equal to 0. Otherwise an F_ULOCK request will attempt to unlock only the requested section.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock the locked region of another process. This implementation detects that sleeping until a locked region is unlocked would cause a deadlock and fails with an EDEADLK error.

lockf(), fcntl(2) and flock(2) locks may be safely used concurrently.

Blocking on a section is interrupted by any signal.

RETURN VALUES

If successful, the **lockf**() function returns 0. Otherwise, it returns −1, sets *errno* to indicate an error, and existing locks are not changed.

ERRORS

lockf() will fail if:

[EAGAIN] The argument *function* is F_TLOCK or F_TEST and the section is already locked by another process.

[EBADF] The argument *filedes* is not a valid open file descriptor.

The argument *function* is F_LOCK or F_TLOCK, and *filedes* is not a valid file descriptor open for writing.

[EDEADLK] The argument *function* is F_LOCK and a deadlock is detected.

[EINTR] The argument *function* is F_LOCK and **lockf**() was interrupted by the delivery of a signal.

[EINVAL] The argument *function* is not one of F_ULOCK, F_LOCK, F_TLOCK or F_TEST.

The argument *filedes* refers to a file that does not support locking.

[ENOLCK] The argument *function* is F_ULOCK, F_LOCK or F_TLOCK, and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

SEE ALSO

fcntl(2), flock(2)

STANDARDS

The **lockf**() function conforms to X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”).

NAME

login, **logout**, **logwtmp** — login utility functions

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

void
login(struct utmp *ut);

int
logout(const char *line);

void
logwtmp(const char *line, const char *name, const char *host);
```

DESCRIPTION

The **login**(), **logout**(), and **logwtmp**() functions operate on the database of current users in /var/run/utmp and on the logfile /var/log/wtmp of logins and logouts.

The **login**() function updates the /var/run/utmp and /var/log/wtmp files with user information contained in *ut*.

The **logout**() function removes the entry from /var/run/utmp corresponding to the device *line*.

The **logwtmp**() function adds an entry to /var/log/wtmp. Since **login**() will add the appropriate entry for /var/log/wtmp during a login, **logwtmp**() is usually used for logouts.

RETURN VALUES

logout() returns non-zero if it was able to find and delete an entry for *line*, and zero if there is no entry for *line* in /var/run/utmp.

FILES

/dev/*
/etc/ttys
/var/run/utmp
/var/log/wtmp

SEE ALSO

utmp(5)

NAME

login_getclass, **login_getcapbool**, **login_getcapnum**, **login_getcapsize**, **login_getcapstr**, **login_getcaptime**, **login_close**, **setclasscontext**, **setusercontext** — query login.conf database about a user class

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <sys/types.h>
#include <login_cap.h>

login_cap_t *
login_getclass(char *class);

int
login_getcapbool(login_cap_t *lc, const char *cap, u_int def);

quad_t
login_getcapnum(login_cap_t *lc, const char *cap, quad_t def, quad_t err);

quad_t
login_getcapsize(login_cap_t *lc, const char *cap, quad_t def, quad_t err);

char *
login_getcapstr(login_cap_t *lc, const char *cap, char *def, char *err);

quad_t
login_getcaptime(login_cap_t *lc, const char *cap, quad_t def, quad_t err);

void
login_close(login_cap_t *lc);

int
setclasscontext(const char *class, u_int flags);

int
setusercontext(login_cap_t *lc, const struct passwd *pwd, uid_t uid,
    u_int flags);
```

DESCRIPTION

The **login_getclass**() function extracts the entry specified by *class* (or `default` if *class* is NULL or the empty string) from `/etc/login.conf` (see `login.conf`(5)). If the entry is found, a `login_cap_t` pointer is returned. NULL is returned if the user class is not found. When the `login_cap_t` structure is no longer needed, it should be freed by the **login_close**() function.

Once *lc* has been returned by **login_getclass**(), any of the other **login_***() functions may be called.

The **login_getcapnum**(), **login_getcapsize**(), **login_getcapstr**(), and **login_getcaptime**() functions all query the database entry for a field named *cap*. If the field is found, its value is returned. If the field is not found, the value specified by *def* is returned. If an error is encountered while trying to find the field, *err* is returned. See `login.conf`(5) for a discussion of the various textual forms the value may take. The **login_getcapbool**() function is slightly different. It returns *def* if no capabilities were found for this class (typically meaning that the default class was used and the `/etc/login.conf` file is missing). It returns a non-zero value if *cap*, with no value, was found, zero otherwise.

The **setclasscontext**() function takes *class*, the name of a user class, and sets the resources defined by that class according to *flags*. Only the LOGIN_SETPATH, LOGIN_SETPRIORITY, LOGIN_SETRESOURCES, and LOGIN_SETUMASK bits are used. (See **setusercontext**() below). It returns 0 on success and -1 on failure.

The **setusercontext**() function sets the resources according to *flags*. The *lc* argument, if not NULL, contains the class information that should be used. The *pwd* argument, if not NULL, provides information about the user. Both *lc* and *pwd* cannot be NULL. The *uid* argument is used in place of the user id contained in the *pwd* structure when calling setuid(2). The various bits available to be or-ed together to make up *flags* are:

LOGIN_SETGID	Set the group id. Requires the *pwd* field be specified.
LOGIN_SETGROUPS	Set the group membership list by calling initgroups(3). Requires the *pwd* field be specified.
LOGIN_SETGROUP	Set the group id and call initgroups(3). Requires the *pwd* field be specified.
LOGIN_SETLOGIN	Sets the login name by setlogin(2). Requires the *pwd* field be specified.
LOGIN_SETPATH	Sets the PATH environment variable.
LOGIN_SETPRIORITY	Sets the priority by setpriority(2).
LOGIN_SETRESOURCES	Sets the various system resources by setrlimit(2).
LOGIN_SETUMASK	Sets the umask by umask(2).
LOGIN_SETUSER	Sets the user id to *uid* by setuid(2).
LOGIN_SETENV	Sets the environment variables as defined by the setenv keyword, by setenv(3).
LOGIN_SETALL	Sets all of the above.

SEE ALSO

setlogin(2), setpriority(2), setrlimit(2), setuid(2), umask(2), initgroups(3), secure_path(3), login.conf(5)

HISTORY

The **login_getclass** family of functions are largely based on the BSD/OS implementation of same, and appeared in NetBSD 1.5 by kind permission.

CAVEATS

The string returned by **login_getcapstr**() is allocated via malloc(3) when the specified capability is present and thus it is the responsibility of the caller to **free**() this space. However, if the capability was not found or an error occurred and *def* or *err* (whichever is relevant) are non-NULL the returned value is simply what was passed in to **login_getcapstr**(). Therefore it is not possible to blindly **free**() the return value without first checking it against *def* and *err*.

The same warnings set forth in setlogin(2) apply to **setusercontext**() when the LOGIN_SETLOGIN flag is used. Specifically, changing the login name affects all processes in the current session, not just the current process. See setlogin(2) for more information.

NAME

loginx, **logoutx**, **logwtmpx** — login utility functions

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

void
loginx(const struct utmpx *ut);

int
logoutx(const char *line, int status, int type);

void
logwtmpx(const char *line, const char *name, const char *host, int status,
    int type);
```

DESCRIPTION

The **loginx**(), **logoutx**(), and **logwtmpx**() operate on the utmpx(5) database of currently logged in users, and the wtmpx(5) database of logins and logouts.

The **loginx**() function updates the /var/run/utmpx and /var/log/wtmpx databases with the information from *ut*.

logoutx() updates the entry corresponding to *line* with the type and status from *type* and *status*.

logwtmpx() writes an entry filled with data from *line*, *name*, *host*, *status*, and *type* to the wtmpx(5) database.

RETURN VALUES

logoutx() returns 1 on success, and 0 if no corresponding entry was found.

SEE ALSO

endutxent(3), utmpx(5)

NAME

llrint, **llrintf**, **lrint**, **lrintf** — convert to integer

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

long long
llrint(double x);

long long
llrintf(float x);

long
lrint(double x);

long
lrintf(float x);
```

DESCRIPTION

The **lrint**() function returns the integer nearest to its argument *x* according to the current rounding mode.

The **llrint**(), **llrintf**(), and **lrintf**() functions differ from **lrint**() only in their input and output types.

RETURN VALUES

The **llrint**, **llrintf**, **lrint**, and **lrintf** functions return the integer nearest to their argument *x* according to the current rounding mode. If the rounded result is too large to be represented as a *long long* or *long* value, respectively, the return value is undefined.

SEE ALSO

math(3), rint(3), round(3)

STANDARDS

The **llrint**(), **llrintf**(), **lrint**(), and **lrintf**() functions conform to ISO/IEC 9899:1999 (“ISO C99”).

NAME

lsearch, **lfind** — linear searching routines

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <search.h>

void *
lsearch(const void *key, void *base, size_t *nelp, size_t width,
    int (*compar)(const void *, const void *));

void *
lfind(const void *key, const void *base, size_t *nelp, size_t width,
    int (*compar)(const void *, const void *));
```

DESCRIPTION

The functions **lsearch**(), and **lfind**() provide basic linear searching functionality.

base is the pointer to the beginning of an array. The argument *nelp* is the current number of elements in the array, where each element is *width* bytes long. The *compar* argument points to a function which compares its two arguments and returns zero if they are matching, and non-zero otherwise.

The **lsearch**() and **lfind**() functions return a pointer into the array referenced by *base* where *key* is located. If *key* does not exist, **lfind**() will return a null pointer and **lsearch**() will add it to the array. When an element is added to the array by **lsearch**() the location referenced by the argument *nelp* is incremented by one.

SEE ALSO

bsearch(3), db(3)

STANDARDS

The **lsearch**() and **lfind**() functions conform to IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

lwres − introduction to the lightweight resolver library

SYNOPSIS

#include <lwres/lwres.h>

DESCRIPTION

The BIND 9 lightweight resolver library is a simple, name service independent stub resolver library. It provides hostname−to−address and address−to−hostname lookup services to applications by transmitting lookup requests to a resolver daemon **lwresd** running on the local host. The resover daemon performs the lookup using the DNS or possibly other name service protocols, and returns the results to the application through the library. The library and resolver daemon communicate using a simple UDP−based protocol.

OVERVIEW

The lwresd library implements multiple name service APIs. The standard **gethostbyname()**, **gethostbyaddr()**, **gethostbyname_r()**, **gethostbyaddr_r()**, **getaddrinfo()**, **getipnodebyname()**, and **getipnodebyaddr()** functions are all supported. To allow the lwres library to coexist with system libraries that define functions of the same name, the library defines these functions with names prefixed by lwres_. To define the standard names, applications must include the header file *<lwres/netdb.h>* which contains macro definitions mapping the standard function names into lwres_ prefixed ones. Operating system vendors who integrate the lwres library into their base distributions should rename the functions in the library proper so that the renaming macros are not needed.

The library also provides a native API consisting of the functions **lwres_getaddrsbyname()** and **lwres_getnamebyaddr()**. These may be called by applications that require more detailed control over the lookup process than the standard functions provide.

In addition to these name service independent address lookup functions, the library implements a new, experimental API for looking up arbitrary DNS resource records, using the **lwres_getaddrsbyname()** function.

Finally, there is a low−level API for converting lookup requests and responses to and from raw lwres protocol packets. This API can be used by clients requiring nonblocking operation, and is also used when implementing the server side of the lwres protocol, for example in the **lwresd** resolver daemon. The use of this low−level API in clients and servers is outlined in the following sections.

CLIENT−SIDE LOW−LEVEL API CALL FLOW

When a client program wishes to make an lwres request using the native low−level API, it typically performs the following sequence of actions.

(1) Allocate or use an existing **lwres_packet_t**, called *pkt* below.

(2) Set pkt.recvlength to the maximum length we will accept. This is done so the receiver of our packets knows how large our receive buffer is. The "default" is a constant in *lwres.h*: **LWRES_RECVLENGTH = 4096**.

(3) Set pkt.serial to a unique serial number. This value is echoed back to the application by the remote server.

(4) Set pkt.pktflags. Usually this is set to 0.

(5) Set pkt.result to 0.

(6) Call **lwres_*request_render()**, or marshall in the data using the primitives such as **lwres_packet_render()** and storing the packet data.

(7) Transmit the resulting buffer.

(8) Call **lwres_*response_parse()** to parse any packets received.

(9) Verify that the opcode and serial match a request, and process the packet specific information contained in the body.

SERVER−SIDE LOW−LEVEL API CALL FLOW

When implementing the server side of the lightweight resolver protocol using the lwres library, a sequence of actions like the following is typically involved in processing each request packet.

Note that the same **lwres_packet_t** is used in both the **_parse()** and **_render()** calls, with only a few modifications made to the packet header's contents between uses. This method is recommended as it keeps the serial, opcode, and other fields correct.

(1) When a packet is received, call **lwres_*request_parse()** to unmarshall it. This returns a **lwres_packet_t** (also called *pkt*, below) as well as a data specific type, such as **lwres_gabnrequest_t**.

(2) Process the request in the data specific type.

(3) Set the pkt.result, pkt.recvlength as above. All other fields can be left untouched since they were filled in by the ***_parse()** call above. If using **lwres_*response_render()**, pkt.pktflags will be set up properly. Otherwise, the **LWRES_LWPACKETFLAG_RESPONSE** bit should be set.

(4) Call the data specific rendering function, such as **lwres_gabnresponse_render()**.

(5) Send the resulting packet to the client.

SEE ALSO

lwres_gethostent(3), **lwres_getipnode**(3), **lwres_getnameinfo**(3), **lwres_noop**(3), **lwres_gabn**(3), **lwres_gnba**(3), **lwres_context**(3), **lwres_config**(3), **resolver**(5), **lwresd**(8).

COPYRIGHT

NAME

lwres_buffer_init, lwres_buffer_invalidate, lwres_buffer_add, lwres_buffer_subtract, lwres_buffer_clear, lwres_buffer_first, lwres_buffer_forward, lwres_buffer_back, lwres_buffer_getuint8, lwres_buffer_putuint8, lwres_buffer_getuint16, lwres_buffer_putuint16, lwres_buffer_getuint32, lwres_buffer_putuint32, lwres_buffer_putmem, lwres_buffer_getmem − lightweight resolver buffer management

SYNOPSIS

#include <lwres/lwbuffer.h>

void lwres_buffer_init(lwres_buffer_t *_b_, void *_base_, unsigned int _length_);

void lwres_buffer_invalidate(lwres_buffer_t *_b_);

void lwres_buffer_add(lwres_buffer_t *_b_, unsigned int _n_);

void lwres_buffer_subtract(lwres_buffer_t *_b_, unsigned int _n_);

void lwres_buffer_clear(lwres_buffer_t *_b_);

void lwres_buffer_first(lwres_buffer_t *_b_);

void lwres_buffer_forward(lwres_buffer_t *_b_, unsigned int _n_);

void lwres_buffer_back(lwres_buffer_t *_b_, unsigned int _n_);

lwres_uint8_t lwres_buffer_getuint8(lwres_buffer_t *_b_);

void lwres_buffer_putuint8(lwres_buffer_t *_b_, lwres_uint8_t _val_);

lwres_uint16_t lwres_buffer_getuint16(lwres_buffer_t *_b_);

void lwres_buffer_putuint16(lwres_buffer_t *_b_, lwres_uint16_t _val_);

lwres_uint32_t lwres_buffer_getuint32(lwres_buffer_t *_b_);

void lwres_buffer_putuint32(lwres_buffer_t *_b_, lwres_uint32_t _val_);

void lwres_buffer_putmem(lwres_buffer_t *_b_, const unsigned char *_base_, unsigned int _length_);

void lwres_buffer_getmem(lwres_buffer_t *_b_, unsigned char *_base_, unsigned int _length_);

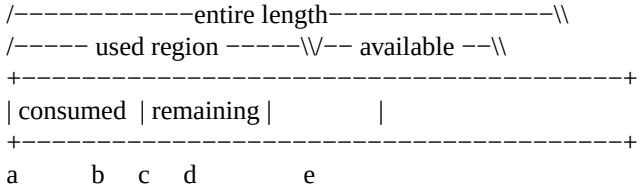
DESCRIPTION

These functions provide bounds checked access to a region of memory where data is being read or written. They are based on, and similar to, the isc_buffer_ functions in the ISC library.

A buffer is a region of memory, together with a set of related subregions. The *used region* and the *available* region are disjoint, and their union is the buffer's region. The used region extends from the beginning of the buffer region to the last used byte. The available region extends from one byte greater than the last used byte to the end of the buffer's region. The size of the used region can be changed using various buffer commands. Initially, the used region is empty.

The used region is further subdivided into two disjoint regions: the *consumed region* and the *remaining region*. The union of these two regions is the used region. The consumed region extends from the beginning of the used region to the byte before the *current* offset (if any). The *remaining* region the current pointer to the end of the used region. The size of the consumed region can be changed using various buffer commands. Initially, the consumed region is empty.

The *active region* is an (optional) subregion of the remaining region. It extends from the current offset to an offset in the remaining region. Initially, the active region is empty. If the current offset advances beyond the chosen offset, the active region will also be empty.



a == base of buffer.
b == current pointer. Can be anywhere between a and d.
c == active pointer. Meaningful between b and d.
d == used pointer.
e == length of buffer.

a−e == entire length of buffer.
a−d == used region.
a−b == consumed region.
b−d == remaining region.
b−c == optional active region.

lwres_buffer_init() initializes the **lwres_buffer_t** **b* and assocates it with the memory region of size *length* bytes starting at location *base.*

lwres_buffer_invalidate() marks the buffer **b* as invalid. Invalidating a buffer after use is not required, but makes it possible to catch its possible accidental use.

The functions **lwres_buffer_add()** and **lwres_buffer_subtract()** respectively increase and decrease the used space in buffer **b* by *n* bytes. **lwres_buffer_add()** checks for buffer overflow and **lwres_buffer_subtract()** checks for underflow. These functions do not allocate or deallocate memory. They just change the value of used.

A buffer is re−initialised by **lwres_buffer_clear()**. The function sets used, current and active to zero.

lwres_buffer_first makes the consumed region of buffer **p* empty by setting current to zero (the start of the buffer).

lwres_buffer_forward() increases the consumed region of buffer **b* by *n* bytes, checking for overflow. Similarly, **lwres_buffer_back()** decreases buffer *b*'s consumed region by *n* bytes and checks for underflow.

lwres_buffer_getuint8() reads an unsigned 8−bit integer from **b* and returns it. **lwres_buffer_putuint8()** writes the unsigned 8−bit integer *val* to buffer **b*.

lwres_buffer_getuint16() and **lwres_buffer_getuint32()** are identical to **lwres_buffer_putuint8()** except that they respectively read an unsigned 16−bit or 32−bit integer in network byte order from *b*. Similarly, **lwres_buffer_putuint16()** and **lwres_buffer_putuint32()** writes the unsigned 16−bit or 32−bit integer *val* to buffer *b*, in network byte order.

Arbitrary amounts of data are read or written from a lightweight resolver buffer with **lwres_buffer_getmem()** and **lwres_buffer_putmem()** respectively. **lwres_buffer_putmem()** copies *length* bytes of memory at *base* to *b*. Conversely, **lwres_buffer_getmem()** copies *length* bytes of memory from *b* to *base*.

COPYRIGHT

NAME

lwres_conf_init, lwres_conf_clear, lwres_conf_parse, lwres_conf_print, lwres_conf_get − lightweight resolver configuration

SYNOPSIS

#include <lwres/lwres.h>

void lwres_conf_init(lwres_context_t *_ctx_);

void lwres_conf_clear(lwres_context_t *_ctx_);

lwres_result_t lwres_conf_parse(lwres_context_t *_ctx_, const char * _filename_);

lwres_result_t lwres_conf_print(lwres_context_t *_ctx_, FILE * _fp_);

lwres_conf_t * lwres_conf_get(lwres_context_t *_ctx_);

DESCRIPTION

lwres_conf_init() creates an empty **lwres_conf_t** structure for lightweight resolver context *ctx*.

lwres_conf_clear() frees up all the internal memory used by that **lwres_conf_t** structure in resolver context *ctx*.

lwres_conf_parse() opens the file *filename* and parses it to initialise the resolver context *ctx*'s **lwres_conf_t** structure.

lwres_conf_print() prints the **lwres_conf_t** structure for resolver context *ctx* to the **FILE** *fp*.

RETURN VALUES

lwres_conf_parse() returns **LWRES_R_SUCCESS** if it successfully read and parsed *filename*. It returns **LWRES_R_FAILURE** if *filename* could not be opened or contained incorrect resolver statements.

lwres_conf_print() returns **LWRES_R_SUCCESS** unless an error occurred when converting the network addresses to a numeric host address string. If this happens, the function returns **LWRES_R_FAILURE**.

SEE ALSO

stdio(3), **resolver**(5).

FILES

/etc/resolv.conf

COPYRIGHT

NAME

lwres_context_create, lwres_context_destroy, lwres_context_nextserial, lwres_context_initserial, lwres_context_freemem, lwres_context_allocmem, lwres_context_sendrecv − lightweight resolver context management

SYNOPSIS

#include <lwres/lwres.h>

lwres_result_t lwres_context_create(lwres_context_t *****contextp*, void ****arg***,**
lwres_malloc_t *malloc_function*,
lwres_free_t *free_function*);

lwres_result_t lwres_context_destroy(lwres_context_t *****contextp*);**

void lwres_context_initserial(lwres_context_t ****ctx*, lwres_uint32_t** *serial*);

lwres_uint32_t lwres_context_nextserial(lwres_context_t ****ctx*);**

void lwres_context_freemem(lwres_context_t ****ctx*, void ****mem***, size_t** *len*);

void lwres_context_allocmem(lwres_context_t ****ctx*, size_t** *len*);

void * lwres_context_sendrecv(lwres_context_t ****ctx*, void ****sendbase***, int** *sendlen*, **void ****recvbase*,
int *recvlen*, **int ****recvd_len*);

DESCRIPTION

lwres_context_create() creates a **lwres_context_t** structure for use in lightweight resolver operations. It holds a socket and other data needed for communicating with a resolver daemon. The new **lwres_context_t** is returned through *contextp*, a pointer to a **lwres_context_t** pointer. This **lwres_context_t** pointer must initially be NULL, and is modified to point to the newly created **lwres_context_t**.

When the lightweight resolver needs to perform dynamic memory allocation, it will call *malloc_function* to allocate memory and *free_function* to free it. If *malloc_function* and *free_function* are NULL, memory is allocated using **malloc**(3). and **free**(3). It is not permitted to have a NULL *malloc_function* and a non−NULL *free_function* or vice versa. *arg* is passed as the first parameter to the memory allocation functions. If *malloc_function* and *free_function* are NULL, *arg* is unused and should be passed as NULL.

Once memory for the structure has been allocated, it is initialized using **lwres_conf_init**(3) and returned via **contextp*.

lwres_context_destroy() destroys a **lwres_context_t**, closing its socket. *contextp* is a pointer to a pointer to the context that is to be destroyed. The pointer will be set to NULL when the context has been destroyed.

The context holds a serial number that is used to identify resolver request packets and associate responses with the corresponding requests. This serial number is controlled using **lwres_context_initserial()** and **lwres_context_nextserial()**. **lwres_context_initserial()** sets the serial number for context **ctx* to *serial*. **lwres_context_nextserial()** increments the serial number and returns the previous value.

Memory for a lightweight resolver context is allocated and freed using **lwres_context_allocmem()** and **lwres_context_freemem()**. These use whatever allocations were defined when the context was created with **lwres_context_create()**. **lwres_context_allocmem()** allocates *len* bytes of memory and if successful returns a pointer to the allocated storage. **lwres_context_freemem()** frees *len* bytes of space starting at location *mem*.

lwres_context_sendrecv() performs I/O for the context *ctx*. Data are read and written from the context's socket. It writes data from *sendbase* — typically a lightweight resolver query packet — and waits for a reply which is copied to the receive buffer at *recvbase*. The number of bytes that were written to this receive buffer is returned in **recvd_len*.

RETURN VALUES

lwres_context_create() returns **LWRES_R_NOMEMORY** if memory for the **struct lwres_context** could not be allocated, **LWRES_R_SUCCESS** otherwise.

Successful calls to the memory allocator **lwres_context_allocmem()** return a pointer to the start of the allocated space. It returns NULL if memory could not be allocated.

LWRES_R_SUCCESS is returned when **lwres_context_sendrecv()** completes successfully. **LWRES_R_IOERROR** is returned if an I/O error occurs and **LWRES_R_TIMEOUT** is returned if **lwres_context_sendrecv()** times out waiting for a response.

SEE ALSO

lwres_conf_init(3), **malloc**(3), **free**(3).

COPYRIGHT

NAME

lwres_gabnrequest_render, lwres_gabnresponse_render, lwres_gabnrequest_parse, lwres_gabnresponse_parse, lwres_gabnresponse_free, lwres_gabnrequest_free − lightweight resolver getaddrbyname message handling

SYNOPSIS

#include <lwres/lwres.h>

lwres_result_t lwres_gabnrequest_render(lwres_context_t ****ctx*, lwres_gabnrequest_t *****req*****,**
lwres_lwpacket_t *****pkt***, lwres_buffer_t *****b*****);**

lwres_result_t lwres_gabnresponse_render(lwres_context_t *****ctx***, lwres_gabnresponse_t *****req*****,**
lwres_lwpacket_t *****pkt***, lwres_buffer_t *****b*****);**

lwres_result_t lwres_gabnrequest_parse(lwres_context_t *****ctx***, lwres_buffer_t *****b*****,**
lwres_lwpacket_t * *pkt*****, lwres_gabnrequest_t ******structp*****);**

lwres_result_t lwres_gabnresponse_parse(lwres_context_t *****ctx***, lwres_buffer_t *****b*****,**
lwres_lwpacket_t *****pkt***,**
lwres_gabnresponse_t ******structp***);**

void lwres_gabnresponse_free(lwres_context_t *****ctx***, lwres_gabnresponse_t ******structp*****);**

void lwres_gabnrequest_free(lwres_context_t *****ctx***, lwres_gabnrequest_t ******structp*****);**

DESCRIPTION

These are low−level routines for creating and parsing lightweight resolver name−to−address lookup request and response messages.

There are four main functions for the getaddrbyname opcode. One render function converts a getaddrbyname request structure — **lwres_gabnrequest_t** — to the lighweight resolver's canonical format. It is complemented by a parse function that converts a packet in this canonical format to a getaddrbyname request structure. Another render function converts the getaddrbyname response structure — **lwres_gabnresponse_t** — to the canonical format. This is complemented by a parse function which converts a packet in canonical format to a getaddrbyname response structure.

These structures are defined in *<lwres/lwres.h>*. They are shown below.

```
#define LWRES_OPCODE_GETADDRSBYNAME     0x00010001U


typedef struct lwres_addr lwres_addr_t;
typedef LWRES_LIST(lwres_addr_t) lwres_addrlist_t;


typedef struct {
        lwres_uint32_t  flags;
        lwres_uint32_t  addrtypes;
        lwres_uint16_t  namelen;
        char            *name;
} lwres_gabnrequest_t;


typedef struct {
        lwres_uint32_t          flags;
        lwres_uint16_t          naliases;
        lwres_uint16_t          naddrs;
        char                    *realname;
        char                    **aliases;
        lwres_uint16_t          realnamelen;
        lwres_uint16_t          *aliaslen;
        lwres_addrlist_t        addrs;
        void                    *base;
```

```
        size_t          baselen;
} lwres_gabnresponse_t;
```

lwres_gabnrequest_render() uses resolver context *ctx* to convert getaddrbyname request structure *req* to canonical format. The packet header structure *pkt* is initialised and transferred to buffer *b*. The contents of **req* are then appended to the buffer in canonical format. **lwres_gabnresponse_render()** performs the same task, except it converts a getaddrbyname response structure **lwres_gabnresponse_t** to the lightweight resolver's canonical format.

lwres_gabnrequest_parse() uses context *ctx* to convert the contents of packet *pkt* to a **lwres_gabnrequest_t** structure. Buffer *b* provides space to be used for storing this structure. When the function succeeds, the resulting **lwres_gabnrequest_t** is made available through **structp*. **lwres_gabnresponse_parse()** offers the same semantics as **lwres_gabnrequest_parse()** except it yields a **lwres_gabnresponse_t** structure.

lwres_gabnresponse_free() and **lwres_gabnrequest_free()** release the memory in resolver context *ctx* that was allocated to the **lwres_gabnresponse_t** or **lwres_gabnrequest_t** structures referenced via *structp*. Any memory associated with ancillary buffers and strings for those structures is also discarded.

RETURN VALUES

The getaddrbyname opcode functions **lwres_gabnrequest_render()**, **lwres_gabnresponse_render()** **lwres_gabnrequest_parse()** and **lwres_gabnresponse_parse()** all return **LWRES_R_SUCCESS** on success. They return **LWRES_R_NOMEMORY** if memory allocation fails. **LWRES_R_UNEXPECTEDEND** is returned if the available space in the buffer *b* is too small to accommodate the packet header or the **lwres_gabnrequest_t** and **lwres_gabnresponse_t** structures. **lwres_gabnrequest_parse()** and **lwres_gabnresponse_parse()** will return **LWRES_R_UNEXPECTEDEND** if the buffer is not empty after decoding the received packet. These functions will return **LWRES_R_FAILURE** if pktflags in the packet header structure **lwres_lwpacket_t** indicate that the packet is not a response to an earlier query.

SEE ALSO

lwres_packet(3)

COPYRIGHT

NAME

lwres_gai_strerror − print suitable error string

SYNOPSIS

#include <lwres/netdb.h>

char * gai_strerror(int *ecode***);**

DESCRIPTION

lwres_gai_strerror() returns an error message corresponding to an error code returned by **getaddrinfo()**. The following error codes and their meaning are defined in *include/lwres/netdb.h*.

EAI_ADDRFAMILY
address family for hostname not supported

EAI_AGAIN
temporary failure in name resolution

EAI_BADFLAGS
invalid value for **ai_flags**

EAI_FAIL
non−recoverable failure in name resolution

EAI_FAMILY
ai_family not supported

EAI_MEMORY
memory allocation failure

EAI_NODATA
no address associated with hostname

EAI_NONAME
hostname or servname not provided, or not known

EAI_SERVICE
servname not supported for **ai_socktype**

EAI_SOCKTYPE
ai_socktype not supported

EAI_SYSTEM
system error returned in errno

The message invalid error code is returned if *ecode* is out of range.

ai_flags, **ai_family** and **ai_socktype** are elements of the **struct addrinfo** used by **lwres_getaddrinfo()**.

SEE ALSO

strerror(3), **lwres_getaddrinfo**(3), **getaddrinfo**(3), **RFC2133**().

COPYRIGHT

NAME

lwres_getaddrinfo, lwres_freeaddrinfo − socket address structure to host and service name

SYNOPSIS

#include <lwres/netdb.h>

int lwres_getaddrinfo(const char ****hostname*, const char ****servname***, const struct addrinfo ****hints***, struct addrinfo *****res***);**

void lwres_freeaddrinfo(struct addrinfo ****ai*);**

If the operating system does not provide a **struct addrinfo**, the following structure is used:

```
struct  addrinfo {
        int             ai_flags;       /* AI_PASSIVE, AI_CANONNAME */
        int             ai_family;      /* PF_xxx */
        int             ai_socktype;    /* SOCK_xxx */
        int             ai_protocol;    /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
        size_t          ai_addrlen;     /* length of ai_addr */
        char            *ai_canonname;  /* canonical name for hostname */
        struct sockaddr *ai_addr;       /* binary address */
        struct addrinfo *ai_next;       /* next structure in linked list */
};
```

DESCRIPTION

lwres_getaddrinfo() is used to get a list of IP addresses and port numbers for host *hostname* and service *servname*. The function is the lightweight resolver's implementation of **getaddrinfo()** as defined in RFC2133. *hostname* and *servname* are pointers to null−terminated strings or **NULL**. *hostname* is either a host name or a numeric host address string: a dotted decimal IPv4 address or an IPv6 address. *servname* is either a decimal port number or a service name as listed in */etc/services*.

hints is an optional pointer to a **struct addrinfo**. This structure can be used to provide hints concerning the type of socket that the caller supports or wishes to use. The caller can supply the following structure elements in **hints*:

ai_family

The protocol family that should be used. When **ai_family** is set to **PF_UNSPEC**, it means the caller will accept any protocol family supported by the operating system.

ai_socktype

denotes the type of socket — **SOCK_STREAM**, **SOCK_DGRAM** or **SOCK_RAW** — that is wanted. When **ai_socktype** is zero the caller will accept any socket type.

ai_protocol

indicates which transport protocol is wanted: IPPROTO_UDP or IPPROTO_TCP. If **ai_protocol** is zero the caller will accept any protocol.

ai_flags

Flag bits. If the **AI_CANONNAME** bit is set, a successful call to **lwres_getaddrinfo()** will return a null−terminated string containing the canonical name of the specified hostname in **ai_canonname** of the first **addrinfo** structure returned. Setting the **AI_PASSIVE** bit indicates that the returned socket address structure is intended for used in a call to **bind**(2). In this case, if the hostname argument is a **NULL** pointer, then the IP address portion of the socket address structure will be set to **INADDR_ANY** for an IPv4 address or **IN6ADDR_ANY_INIT** for an IPv6 address.

When **ai_flags** does not set the **AI_PASSIVE** bit, the returned socket address structure will be ready for use in a call to **connect**(2) for a connection−oriented protocol or **connect**(2), **sendto**(2), or **sendmsg**(2) if a connectionless protocol was chosen. The IP address portion of the socket address structure will be set to the loopback address if *hostname* is a **NULL** pointer and **AI_PASSIVE** is not set in **ai_flags**.

If **ai_flags** is set to **AI_NUMERICHOST** it indicates that *hostname* should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.

All other elements of the **struct addrinfo** passed via *hints* must be zero.

A *hints* of **NULL** is treated as if the caller provided a **struct addrinfo** initialized to zero with **ai_family**set to **PF_UNSPEC**.

After a successful call to **lwres_getaddrinfo()**, **res* is a pointer to a linked list of one or more **addrinfo** structures. Each **struct addrinfo** in this list cn be processed by following the **ai_next** pointer, until a **NULL** pointer is encountered. The three members **ai_family**, **ai_socktype**, and **ai_protocol** in each returned **addrinfo** structure contain the corresponding arguments for a call to **socket**(2). For each **addrinfo** structure in the list, the **ai_addr** member points to a filled−in socket address structure of length **ai_addrlen**.

All of the information returned by **lwres_getaddrinfo()** is dynamically allocated: the addrinfo structures, and the socket address structures and canonical host name strings pointed to by the **addrinfo**structures. Memory allocated for the dynamically allocated structures created by a successful call to **lwres_getaddrinfo()** is released by **lwres_freeaddrinfo()**. *ai* is a pointer to a **struct addrinfo** created by a call to **lwres_getaddrinfo()**.

RETURN VALUES

lwres_getaddrinfo() returns zero on success or one of the error codes listed in **gai_strerror**(3) if an error occurs. If both *hostname* and *servname* are **NULL lwres_getaddrinfo()** returns **EAI_NONAME**.

SEE ALSO

lwres(3), **lwres_getaddrinfo**(3), **lwres_freeaddrinfo**(3), **lwres_gai_strerror**(3), **RFC2133**(), **getservbyname**(3), **bind**(2), **connect**(2), **sendto**(2), **sendmsg**(2), **socket**(2).

COPYRIGHT

NAME

lwres_gethostbyname, lwres_gethostbyname2, lwres_gethostbyaddr, lwres_gethostent, lwres_sethostent, lwres_endhostent, lwres_gethostbyname_r, lwres_gethostbyaddr_r, lwres_gethostent_r, lwres_sethostent_r, lwres_endhostent_r − lightweight resolver get network host entry

SYNOPSIS

#include <lwres/netdb.h>

struct hostent * lwres_gethostbyname(const char * *name***);**

struct hostent * lwres_gethostbyname2(const char * *name***, int** *af***);**

struct hostent * lwres_gethostbyaddr(const char * *addr***, int** *len***, int** *type***);**

struct hostent * lwres_gethostent(void);

void lwres_sethostent(int *stayopen***);**

void lwres_endhostent(void);

struct hostent * lwres_gethostbyname_r(const char * *name***, struct hostent *** *resbuf***, char *** *buf***, int** *buflen***, int *** *error***);**

struct hostent * lwres_gethostbyaddr_r(const char * *addr***, int** *len***, int** *type***, struct hostent *** *resbuf***, char *** *buf***, int** *buflen***, int *** *error***);**

struct hostent * lwres_gethostent_r(struct hostent * *resbuf***, char *** *buf***, int** *buflen***, int *** *error***);**

void lwres_sethostent_r(int *stayopen***);**

void lwres_endhostent_r(void);

DESCRIPTION

These functions provide hostname−to−address and address−to−hostname lookups by means of the lightweight resolver. They are similar to the standard **gethostent**(3) functions provided by most operating systems. They use a **struct hostent** which is usually defined in <*namedb.h*>.

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses from name server */
};
#define h_addr  h_addr_list[0]  /* address, for backward compatibility */
```

The members of this structure are:

h_name
The official (canonical) name of the host.

h_aliases
A NULL−terminated array of alternate names (nicknames) for the host.

h_addrtype
The type of address being returned — **PF_INET** or **PF_INET6**.

h_length
The length of the address in bytes.

h_addr_list
A **NULL** terminated array of network addresses for the host. Host addresses are returned in network byte order.

For backward compatibility with very old software, **h_addr** is the first address in **h_addr_list.**

lwres_gethostent(), **lwres_sethostent()**, **lwres_endhostent()**, **lwres_gethostent_r()**, **lwres_sethostent_r()**

and **lwres_endhostent_r()** provide iteration over the known host entries on systems that provide such functionality through facilities like */etc/hosts* or NIS. The lightweight resolver does not currently implement these functions; it only provides them as stub functions that always return failure.

lwres_gethostbyname() and **lwres_gethostbyname2()** look up the hostname *name*. **lwres_gethostbyname()** always looks for an IPv4 address while **lwres_gethostbyname2()** looks for an address of protocol family *af*: either **PF_INET** or **PF_INET6** — IPv4 or IPV6 addresses respectively. Successful calls of the functions return a **struct hostent**for the name that was looked up. **NULL** is returned if the lookups by **lwres_gethostbyname()** or **lwres_gethostbyname2()** fail.

Reverse lookups of addresses are performed by **lwres_gethostbyaddr()**. *addr* is an address of length *len* bytes and protocol family *type* — **PF_INET** or **PF_INET6**. **lwres_gethostbyname_r()** is a thread−safe function for forward lookups. If an error occurs, an error code is returned in **error*. *resbuf* is a pointer to a **struct hostent** which is initialised by a successful call to **lwres_gethostbyname_r()**. *buf* is a buffer of length *len* bytes which is used to store the **h_name**, **h_aliases**, and **h_addr_list** elements of the **struct hostent** returned in *resbuf*. Successful calls to **lwres_gethostbyname_r()** return *resbuf*, which is a pointer to the **struct hostent** it created.

lwres_gethostbyaddr_r() is a thread−safe function that performs a reverse lookup of address *addr* which is *len* bytes long and is of protocol family *type* — **PF_INET** or **PF_INET6**. If an error occurs, the error code is returned in **error*. The other function parameters are identical to those in **lwres_gethostbyname_r()**. *resbuf* is a pointer to a **struct hostent** which is initialised by a successful call to **lwres_gethostbyaddr_r()**. *buf* is a buffer of length *len* bytes which is used to store the **h_name**, **h_aliases**, and **h_addr_list** elements of the **struct hostent** returned in *resbuf*. Successful calls to **lwres_gethostbyaddr_r()** return *resbuf*, which is a pointer to the **struct hostent()** it created.

RETURN VALUES

The functions **lwres_gethostbyname()**, **lwres_gethostbyname2()**, **lwres_gethostbyaddr()**, and **lwres_gethostent()** return NULL to indicate an error. In this case the global variable **lwres_h_errno** will contain one of the following error codes defined in <*lwres/netdb.h*>:

HOST_NOT_FOUND
The host or address was not found.

TRY_AGAIN
A recoverable error occurred, e.g., a timeout. Retrying the lookup may succeed.

NO_RECOVERY
A non−recoverable error occurred.

NO_DATA
The name exists, but has no address information associated with it (or vice versa in the case of a reverse lookup). The code NO_ADDRESS is accepted as a synonym for NO_DATA for backwards compatibility.

lwres_hstrerror(3) translates these error codes to suitable error messages.

lwres_gethostent() and **lwres_gethostent_r()** always return **NULL**.

Successful calls to **lwres_gethostbyname_r()** and **lwres_gethostbyaddr_r()** return *resbuf*, a pointer to the **struct hostent** that was initialised by these functions. They return **NULL** if the lookups fail or if *buf* was too small to hold the list of addresses and names referenced by the **h_name**, **h_aliases**, and **h_addr_list** elements of the **struct hostent**. If *buf* was too small, both **lwres_gethostbyname_r()** and **lwres_gethostbyaddr_r()** set the global variable **errno** to **ERANGE**.

SEE ALSO

gethostent(3), **lwres_getipnode**(3), **lwres_hstrerror**(3)

BUGS

lwres_gethostbyname(), **lwres_gethostbyname2()**, **lwres_gethostbyaddr()** and **lwres_endhostent()** are not thread safe; they return pointers to static data and provide error codes through a global variable. Thread−safe versions for name and address lookup are provided by **lwres_gethostbyname_r()**, and

lwres_gethostbyaddr_r() respectively.

The resolver daemon does not currently support any non−DNS name services such as */etc/hosts* or **NIS**, consequently the above functions don’t, either.

COPYRIGHT

NAME

lwres_getipnodebyname, lwres_getipnodebyaddr, lwres_freehostent − lightweight resolver nodename / address translation API

SYNOPSIS

#include <lwres/netdb.h>

struct hostent * lwres_getipnodebyname(const char * *name***, int** *af***, int** *flags***, int *** *error_num***);**

struct hostent * lwres_getipnodebyaddr(const void * *src***, size_t** *len***, int** *af***, int *** *error_num***);**

void lwres_freehostent(struct hostent * *he***);**

DESCRIPTION

These functions perform thread safe, protocol independent nodename−to−address and address−to−nodename translation as defined in RFC2553.

They use a **struct hostent** which is defined in *namedb.h*:

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses from name server */
};
#define h_addr  h_addr_list[0]  /* address, for backward compatibility */
```

The members of this structure are:

h_name
The official (canonical) name of the host.

h_aliases
A NULL−terminated array of alternate names (nicknames) for the host.

h_addrtype
The type of address being returned − usually **PF_INET** or **PF_INET6**.

h_length
The length of the address in bytes.

h_addr_list
A **NULL** terminated array of network addresses for the host. Host addresses are returned in network byte order.

lwres_getipnodebyname() looks up addresses of protocol family *af* for the hostname *name*. The *flags* parameter contains ORed flag bits to specify the types of addresses that are searched for, and the types of addresses that are returned. The flag bits are:

AI_V4MAPPED
This is used with an *af* of AF_INET6, and causes IPv4 addresses to be returned as IPv4−mapped IPv6 addresses.

AI_ALL
This is used with an *af* of AF_INET6, and causes all known addresses (IPv6 and IPv4) to be returned. If AI_V4MAPPED is also set, the IPv4 addresses are return as mapped IPv6 addresses.

AI_ADDRCONFIG
Only return an IPv6 or IPv4 address if here is an active network interface of that type. This is not currently implemented in the BIND 9 lightweight resolver, and the flag is ignored.

AI_DEFAULT
This default sets the **AI_V4MAPPED** and **AI_ADDRCONFIG** flag bits.

lwres_getipnodebyaddr() performs a reverse lookup of address *src* which is *len* bytes long. *af* denotes the protocol family, typically **PF_INET** or **PF_INET6**.

lwres_freehostent() releases all the memory associated with the **struct hostent** pointer *he*. Any memory allocated for the **h_name**, **h_addr_list** and **h_aliases** is freed, as is the memory for the **hostent** structure itself.

RETURN VALUES

If an error occurs, **lwres_getipnodebyname()** and **lwres_getipnodebyaddr()** set **error_num* to an appropriate error code and the function returns a **NULL** pointer. The error codes and their meanings are defined in <*lwres/netdb.h*>:

HOST_NOT_FOUND
No such host is known.

NO_ADDRESS
The server recognised the request and the name but no address is available. Another type of request to the name server for the domain might return an answer.

TRY_AGAIN
A temporary and possibly transient error occurred, such as a failure of a server to respond. The request may succeed if retried.

NO_RECOVERY
An unexpected failure occurred, and retrying the request is pointless.

lwres_hstrerror(3) translates these error codes to suitable error messages.

SEE ALSO

RFC2553(), **lwres**(3), **lwres_gethostent**(3), **lwres_getaddrinfo**(3), **lwres_getnameinfo**(3), **lwres_hstrerror**(3).

COPYRIGHT

NAME

lwres_getnameinfo − lightweight resolver socket address structure to hostname and service name

SYNOPSIS

#include <lwres/netdb.h>

int lwres_getnameinfo(const struct sockaddr ****sa*, size_t** *salen***, char *****host***, size_t** *hostlen***, char *****serv***,**
size_t *servlen***, int** *flags***);**

DESCRIPTION

This function is equivalent to the **getnameinfo**(3) function defined in RFC2133. **lwres_getnameinfo()** returns the hostname for the **struct sockaddr** *sa* which is *salen* bytes long. The hostname is of length *hostlen* and is returned via **host.* The maximum length of the hostname is 1025 bytes: **NI_MAXHOST**.

The name of the service associated with the port number in *sa* is returned in **serv.* It is *servlen* bytes long. The maximum length of the service name is **NI_MAXSERV** − 32 bytes.

The *flags* argument sets the following bits:

NI_NOFQDN
A fully qualified domain name is not required for local hosts. The local part of the fully qualified domain name is returned instead.

NI_NUMERICHOST
Return the address in numeric form, as if calling inet_ntop(), instead of a host name.

NI_NAMEREQD
A name is required. If the hostname cannot be found in the DNS and this flag is set, a non−zero error code is returned. If the hostname is not found and the flag is not set, the address is returned in numeric form.

NI_NUMERICSERV
The service name is returned as a digit string representing the port number.

NI_DGRAM
Specifies that the service being looked up is a datagram service, and causes getservbyport() to be called with a second argument of "udp" instead of its default of "tcp". This is required for the few ports (512−514) that have different services for UDP and TCP.

RETURN VALUES

lwres_getnameinfo() returns 0 on success or a non−zero error code if an error occurs.

SEE ALSO

RFC2133(), **getservbyport**(3), **lwres**(3), **lwres_getnameinfo**(3), **lwres_getnamebyaddr**(3). **lwres_net_ntop**(3).

BUGS

RFC2133 fails to define what the nonzero return values of **getnameinfo**(3) are.

COPYRIGHT

NAME

lwres_getrrsetbyname, lwres_freerrset − retrieve DNS records

SYNOPSIS

#include <lwres/netdb.h>

int lwres_getrrsetbyname(const char ****hostname*, unsigned int** *rdclass***, unsigned int** *rdtype***, unsigned int** *flags***, struct rrsetinfo *****res***);**

void lwres_freerrset(struct rrsetinfo ****rrset*);**

The following structures are used:

```
struct  rdatainfo {
        unsigned int            rdi_length;     /* length of data */
        unsigned char           *rdi_data;      /* record data */
};


struct  rrsetinfo {
        unsigned int            rri_flags;      /* RRSET_VALIDATED... */
        unsigned int            rri_rdclass;    /* class number */
        unsigned int            rri_rdtype;     /* RR type number */
        unsigned int            rri_ttl;        /* time to live */
        unsigned int            rri_nrdatas;    /* size of rdatas array */
        unsigned int            rri_nsigs;      /* size of sigs array */
        char                    *rri_name;      /* canonical name */
        struct rdatainfo        *rri_rdatas;    /* individual records */
        struct rdatainfo        *rri_sigs;      /* individual signatures */
};
```

DESCRIPTION

lwres_getrrsetbyname() gets a set of resource records associated with a *hostname*, *class*, and *type*. *hostname* is a pointer a to null−terminated string. The *flags* field is currently unused and must be zero.

After a successful call to **lwres_getrrsetbyname()**, **res* is a pointer to an **rrsetinfo** structure, containing a list of one or more **rdatainfo** structures containing resource records and potentially another list of **rdatainfo** structures containing SIG resource records associated with those records. The members **rri_rdclass** and **rri_rdtype** are copied from the parameters. **rri_ttl** and **rri_name** are properties of the obtained rrset. The resource records contained in **rri_rdatas** and **rri_sigs** are in uncompressed DNS wire format. Properties of the rdataset are represented in the **rri_flags** bitfield. If the RRSET_VALIDATED bit is set, the data has been DNSSEC validated and the signatures verified.

All of the information returned by **lwres_getrrsetbyname()** is dynamically allocated: the **rrsetinfo** and **rdatainfo** structures, and the canonical host name strings pointed to by the **rrsetinfo**structure. Memory allocated for the dynamically allocated structures created by a successful call to **lwres_getrrsetbyname()** is released by **lwres_freerrset()**. *rrset* is a pointer to a **struct rrset** created by a call to **lwres_getrrsetbyname()**.

RETURN VALUES

lwres_getrrsetbyname() returns zero on success, and one of the following error codes if an error occurred:

ERRSET_NONAME
the name does not exist

ERRSET_NODATA
the name exists, but does not have data of the desired type

ERRSET_NOMEMORY
memory could not be allocated

ERRSET_INVAL
a parameter is invalid

ERRSET_FAIL
other failure

SEE ALSO

lwres(3).

COPYRIGHT

NAME

lwres_gnbarequest_render, lwres_gnbaresponse_render, lwres_gnbarequest_parse, lwres_gnbaresponse_parse, lwres_gnbaresponse_free, lwres_gnbarequest_free − lightweight resolver getnamebyaddress message handling

SYNOPSIS

#include <lwres/lwres.h>

lwres_result_t lwres_gnbarequest_render(lwres_context_t ****ctx*, lwres_gnbarequest_t *****req*****, lwres_lwpacket_t *****pkt*****, lwres_buffer_t *****b*****);**

lwres_result_t lwres_gnbaresponse_render(lwres_context_t ****ctx*, lwres_gnbaresponse_t *****req*****, lwres_lwpacket_t *****pkt*****, lwres_buffer_t *****b*****);**

lwres_result_t lwres_gnbarequest_parse(lwres_context_t ****ctx*, lwres_buffer_t *****b*****, lwres_lwpacket_t *****pkt*****, lwres_gnbarequest_t ******structp*****);**

lwres_result_t lwres_gnbaresponse_parse(lwres_context_t ****ctx*, lwres_buffer_t *****b*****, lwres_lwpacket_t *****pkt*****, lwres_gnbaresponse_t ******structp*****);**

void lwres_gnbaresponse_free(lwres_context_t ****ctx*, lwres_gnbaresponse_t ******structp*****);**

void lwres_gnbarequest_free(lwres_context_t ****ctx*, lwres_gnbarequest_t ******structp*****);**

DESCRIPTION

These are low−level routines for creating and parsing lightweight resolver address−to−name lookup request and response messages.

There are four main functions for the getnamebyaddr opcode. One render function converts a getnamebyaddr request structure — **lwres_gnbarequest_t** — to the lightweight resolver's canonical format. It is complemented by a parse function that converts a packet in this canonical format to a getnamebyaddr request structure. Another render function converts the getnamebyaddr response structure — **lwres_gnbaresponse_t** to the canonical format. This is complemented by a parse function which converts a packet in canonical format to a getnamebyaddr response structure.

These structures are defined in *lwres/lwres.h*. They are shown below.

```
#define LWRES_OPCODE_GETNAMEBYADDR      0x00010002U


typedef struct {
        lwres_uint32_t  flags;
        lwres_addr_t    addr;
} lwres_gnbarequest_t;


typedef struct {
        lwres_uint32_t  flags;
        lwres_uint16_t  naliases;
        char            *realname;
        char            **aliases;
        lwres_uint16_t  realnamelen;
        lwres_uint16_t *aliaslen;
        void            *base;
        size_t          baselen;
} lwres_gnbaresponse_t;
```

lwres_gnbarequest_render() uses resolver context *ctx* to convert getnamebyaddr request structure *req* to canonical format. The packet header structure *pkt* is initialised and transferred to buffer *b*. The contents of **req* are then appended to the buffer in canonical format. **lwres_gnbaresponse_render()** performs the same task, except it converts a getnamebyaddr response structure **lwres_gnbaresponse_t** to the lightweight

resolver's canonical format.

lwres_gnbarequest_parse() uses context *ctx* to convert the contents of packet *pkt* to a **lwres_gnbarequest_t** structure. Buffer *b* provides space to be used for storing this structure. When the function succeeds, the resulting **lwres_gnbarequest_t** is made available through **structp*. **lwres_gnbaresponse_parse()** offers the same semantics as **lwres_gnbarequest_parse()** except it yields a **lwres_gnbaresponse_t** structure.

lwres_gnbaresponse_free() and **lwres_gnbarequest_free()** release the memory in resolver context *ctx* that was allocated to the **lwres_gnbaresponse_t** or **lwres_gnbarequest_t** structures referenced via *structp*. Any memory associated with ancillary buffers and strings for those structures is also discarded.

RETURN VALUES

The getnamebyaddr opcode functions **lwres_gnbarequest_render()**, **lwres_gnbaresponse_render()** **lwres_gnbarequest_parse()** and **lwres_gnbaresponse_parse()** all return **LWRES_R_SUCCESS** on success. They return **LWRES_R_NOMEMORY** if memory allocation fails. **LWRES_R_UNEXPECTEDEND** is returned if the available space in the buffer *b* is too small to accommodate the packet header or the **lwres_gnbarequest_t** and **lwres_gnbaresponse_t** structures. **lwres_gnbarequest_parse()** and **lwres_gnbaresponse_parse()** will return **LWRES_R_UNEXPECTEDEND** if the buffer is not empty after decoding the received packet. These functions will return **LWRES_R_FAILURE** if pktflags in the packet header structure **lwres_lwpacket_t** indicate that the packet is not a response to an earlier query.

SEE ALSO

lwres_packet(3).

COPYRIGHT

NAME

lwres_herror, lwres_hstrerror − lightweight resolver error message generation

SYNOPSIS

#include <lwres/netdb.h>

void lwres_herror(const char *_s_);

const char * lwres_hstrerror(int _err_);

DESCRIPTION

lwres_herror() prints the string *s* on **stderr** followed by the string generated by **lwres_hstrerror()** for the error code stored in the global variable **lwres_h_errno**.

lwres_hstrerror() returns an appropriate string for the error code gievn by *err*. The values of the error codes and messages are as follows:

NETDB_SUCCESS
Resolver Error 0 (no error)

HOST_NOT_FOUND
Unknown host

TRY_AGAIN
Host name lookup failure

NO_RECOVERY
Unknown server error

NO_DATA
No address associated with name

RETURN VALUES

The string Unknown resolver error is returned by **lwres_hstrerror()** when the value of **lwres_h_errno** is not a valid error code.

SEE ALSO

herror(3), **lwres_hstrerror**(3).

COPYRIGHT

NAME

lwres_net_ntop − lightweight resolver IP address presentation

SYNOPSIS

#include <lwres/net.h>

const char * lwres_net_ntop(int *af***, const void ****src***, char ****dst***, size_t** *size***);**

DESCRIPTION

lwres_net_ntop() converts an IP address of protocol family *af* — IPv4 or IPv6 — at location *src* from network format to its conventional representation as a string. For IPv4 addresses, that string would be a dotted−decimal. An IPv6 address would be represented in colon notation as described in RFC1884.

The generated string is copied to *dst* provided *size* indicates it is long enough to store the ASCII representation of the address.

RETURN VALUES

If successful, the function returns *dst*: a pointer to a string containing the presentation format of the address. **lwres_net_ntop()** returns **NULL** and sets the global variable **errno** to **EAFNOSUPPORT** if the protocol family given in *af* is not supported.

SEE ALSO

RFC1884(), **inet_ntop**(3), **errno**(3).

COPYRIGHT

NAME

lwres_nooprequest_render, lwres_noopresponse_render, lwres_nooprequest_parse, lwres_noopresponse_parse, lwres_noopresponse_free, lwres_nooprequest_free − lightweight resolver no−op message handling

SYNOPSIS

#include <lwres/lwres.h>

lwres_result_t lwres_nooprequest_render(lwres_context_t ****ctx*, lwres_nooprequest_t *****req*****, lwres_lwpacket_t *****pkt*****, lwres_buffer_t *****b*****);**

lwres_result_t lwres_noopresponse_render(lwres_context_t ****ctx*, lwres_noopresponse_t *****req*****, lwres_lwpacket_t *****pkt*****, lwres_buffer_t *****b*****);**

lwres_result_t lwres_nooprequest_parse(lwres_context_t ****ctx*, lwres_buffer_t *****b*****, lwres_lwpacket_t *****pkt*****, lwres_nooprequest_t ******structp*****);**

lwres_result_t lwres_noopresponse_parse(lwres_context_t ****ctx*, lwres_buffer_t *****b*****, lwres_lwpacket_t *****pkt*****, lwres_noopresponse_t ******structp*****);**

void lwres_noopresponse_free(lwres_context_t ****ctx*, lwres_noopresponse_t ******structp*****);**

void lwres_nooprequest_free(lwres_context_t ****ctx*, lwres_nooprequest_t ******structp*****);**

DESCRIPTION

These are low−level routines for creating and parsing lightweight resolver no−op request and response messages.

The no−op message is analogous to a **ping** packet: a packet is sent to the resolver daemon and is simply echoed back. The opcode is intended to allow a client to determine if the server is operational or not.

There are four main functions for the no−op opcode. One render function converts a no−op request structure — **lwres_nooprequest_t** — to the lighweight resolver's canonical format. It is complemented by a parse function that converts a packet in this canonical format to a no−op request structure. Another render function converts the no−op response structure — **lwres_noopresponse_t** to the canonical format. This is complemented by a parse function which converts a packet in canonical format to a no−op response structure.

These structures are defined in *lwres/lwres.h*. They are shown below.

```
#define LWRES_OPCODE_NOOP       0x00000000U


typedef struct {
        lwres_uint16_t  datalength;
        unsigned char   *data;
} lwres_nooprequest_t;


typedef struct {
        lwres_uint16_t  datalength;
        unsigned char   *data;
} lwres_noopresponse_t;
```

Although the structures have different types, they are identical. This is because the no−op opcode simply echos whatever data was sent: the response is therefore identical to the request.

lwres_nooprequest_render() uses resolver context *ctx* to convert no−op request structure *req* to canonical format. The packet header structure *pkt* is initialised and transferred to buffer *b*. The contents of **req* are then appended to the buffer in canonical format. **lwres_noopresponse_render()** performs the same task, except it converts a no−op response structure **lwres_noopresponse_t** to the lightweight resolver's canonical format.

lwres_nooprequest_parse() uses context *ctx* to convert the contents of packet *pkt* to a **lwres_nooprequest_t** structure. Buffer *b* provides space to be used for storing this structure. When the function succeeds, the resulting **lwres_nooprequest_t** is made available through **structp*. **lwres_noopresponse_parse()** offers the same semantics as **lwres_nooprequest_parse()** except it yields a **lwres_noopresponse_t** structure.

lwres_noopresponse_free() and **lwres_nooprequest_free()** release the memory in resolver context *ctx* that was allocated to the **lwres_noopresponse_t** or **lwres_nooprequest_t** structures referenced via *structp*.

RETURN VALUES

The no−op opcode functions **lwres_nooprequest_render()**, **lwres_noopresponse_render()** **lwres_nooprequest_parse()** and **lwres_noopresponse_parse()** all return **LWRES_R_SUCCESS** on success. They return **LWRES_R_NOMEMORY** if memory allocation fails. **LWRES_R_UNEXPECTEDEND** is returned if the available space in the buffer *b* is too small to accommodate the packet header or the **lwres_nooprequest_t** and **lwres_noopresponse_t** structures. **lwres_nooprequest_parse()** and **lwres_noopresponse_parse()** will return **LWRES_R_UNEXPECTEDEND** if the buffer is not empty after decoding the received packet. These functions will return **LWRES_R_FAILURE** if **pktflags** in the packet header structure **lwres_lwpacket_t** indicate that the packet is not a response to an earlier query.

SEE ALSO

lwres_packet(3)

COPYRIGHT

NAME

lwres_lwpacket_renderheader, lwres_lwpacket_parseheader − lightweight resolver packet handling functions

SYNOPSIS

#include <lwres/lwpacket.h>

lwres_result_t lwres_lwpacket_renderheader(lwres_buffer_t *_b_, lwres_lwpacket_t *_pkt_);

lwres_result_t lwres_lwpacket_parseheader(lwres_buffer_t *_b_, lwres_lwpacket_t *_pkt_);

DESCRIPTION

These functions rely on a **struct lwres_lwpacket** which is defined in *lwres/lwpacket.h*.

```
typedef struct lwres_lwpacket lwres_lwpacket_t;


struct lwres_lwpacket {
        lwres_uint32_t      length;
        lwres_uint16_t      version;
        lwres_uint16_t      pktflags;
        lwres_uint32_t      serial;
        lwres_uint32_t      opcode;
        lwres_uint32_t      result;
        lwres_uint32_t      recvlength;
        lwres_uint16_t      authtype;
        lwres_uint16_t      authlength;
};
```

The elements of this structure are:

length
the overall packet length, including the entire packet header. This field is filled in by the lwres_gabn_*() and lwres_gnba_*() calls.

version
the header format. There is currently only one format, **LWRES_LWPACKETVERSION_0**. This field is filled in by the lwres_gabn_*() and lwres_gnba_*() calls.

pktflags
library−defined flags for this packet: for instance whether the packet is a request or a reply. Flag values can be set, but not defined by the caller. This field is filled in by the application wit the exception of the LWRES_LWPACKETFLAG_RESPONSE bit, which is set by the library in the lwres_gabn_*() and lwres_gnba_*() calls.

serial
is set by the requestor and is returned in all replies. If two or more packets from the same source have the same serial number and are from the same source, they are assumed to be duplicates and the latter ones may be dropped. This field must be set by the application.

opcode
indicates the operation. Opcodes between 0x00000000 and 0x03ffffff are reserved for use by the lightweight resolver library. Opcodes between 0x04000000 and 0xffffffff are application defined. This field is filled in by the lwres_gabn_*() and lwres_gnba_*() calls.

result
is only valid for replies. Results between 0x04000000 and 0xffffffff are application defined. Results between 0x00000000 and 0x03ffffff are reserved for library use. This field is filled in by the lwres_gabn_*() and lwres_gnba_*() calls.

recvlength
is the maximum buffer size that the receiver can handle on requests and the size of the buffer needed to

satisfy a request when the buffer is too large for replies. This field is supplied by the application.

authtype
defines the packet level authentication that is used. Authorisation types between 0x1000 and 0xffff are application defined and types between 0x0000 and 0x0fff are reserved for library use. Currently these are not used and must be zero.

authlen
gives the length of the authentication data. Since packet authentication is currently not used, this must be zero.

The following opcodes are currently defined:

NOOP
Success is always returned and the packet contents are echoed. The lwres_noop_*() functions should be used for this type.

GETADDRSBYNAME
returns all known addresses for a given name. The lwres_gabn_*() functions should be used for this type.

GETNAMEBYADDR
return the hostname for the given address. The lwres_gnba_*() functions should be used for this type.

lwres_lwpacket_renderheader() transfers the contents of lightweight resolver packet structure **lwres_lwpacket_t** **pkt* in network byte order to the lightweight resolver buffer, **b*.

lwres_lwpacket_parseheader() performs the converse operation. It transfers data in network byte order from buffer **b* to resolver packet **pkt*. The contents of the buffer *b* should correspond to a **lwres_lwpacket_t**.

RETURN VALUES

Successful calls to **lwres_lwpacket_renderheader()** and **lwres_lwpacket_parseheader()** return **LWRES_R_SUCCESS**. If there is insufficient space to copy data between the buffer **b* and lightweight resolver packet **pkt* both functions return **LWRES_R_UNEXPECTEDEND**.

COPYRIGHT

NAME

lwres_string_parse, lwres_addr_parse, lwres_getaddrsbyname, lwres_getnamebyaddr − lightweight resolver utility functions

SYNOPSIS

#include <lwres/lwres.h>

lwres_result_t lwres_string_parse(lwres_buffer_t *b, char **c, lwres_uint16_t *len);

lwres_result_t lwres_addr_parse(lwres_buffer_t *b, lwres_addr_t *addr);

lwres_result_t lwres_getaddrsbyname(lwres_context_t *ctx, const char *name,
lwres_uint32_t addrtypes,
lwres_gabnresponse_t **structp);

lwres_result_t lwres_getnamebyaddr(lwres_context_t *ctx, lwres_uint32_t addrtype,
lwres_uint16_t addrlen, const unsigned char *addr,
lwres_gnbaresponse_t **structp);

DESCRIPTION

lwres_string_parse() retrieves a DNS−encoded string starting the current pointer of lightweight resolver buffer *b*: i.e. **b−>current**. When the function returns, the address of the first byte of the encoded string is returned via **c* and the length of that string is given by **len*. The buffer's current pointer is advanced to point at the character following the string length, the encoded string, and the trailing **NULL** character.

lwres_addr_parse() extracts an address from the buffer *b*. The buffer's current pointer **b−>current** is presumed to point at an encoded address: the address preceded by a 32−bit protocol family identifier and a 16−bit length field. The encoded address is copied to **addr−>address** and **addr−>length** indicates the size in bytes of the address that was copied. **b−>current** is advanced to point at the next byte of available data in the buffer following the encoded address.

lwres_getaddrsbyname() and **lwres_getnamebyaddr()** use the **lwres_gnbaresponse_t** structure defined below:

```
typedef struct {
        lwres_uint32_t          flags;
        lwres_uint16_t          naliases;
        lwres_uint16_t          naddrs;
        char                    *realname;
        char                    **aliases;
        lwres_uint16_t          realnamelen;
        lwres_uint16_t          *aliaslen;
        lwres_addrlist_t        addrs;
        void                    *base;
        size_t                  baselen;
} lwres_gabnresponse_t;
```

The contents of this structure are not manipulated directly but they are controlled through the **lwres_gabn**(3) functions.

The lightweight resolver uses **lwres_getaddrsbyname()** to perform foward lookups. Hostname *name* is looked up using the resolver context *ctx* for memory allocation. *addrtypes* is a bitmask indicating which type of addresses are to be looked up. Current values for this bitmask are **LWRES_ADDRTYPE_V4** for IPv4 addresses and **LWRES_ADDRTYPE_V6** for IPv6 addresses. Results of the lookup are returned in **structp*.

lwres_getnamebyaddr() performs reverse lookups. Resolver context *ctx* is used for memory allocation. The address type is indicated by *addrtype*: **LWRES_ADDRTYPE_V4** or **LWRES_ADDRTYPE_V6**. The address to be looked up is given by *addr* and its length is *addrlen* bytes. The result of the function call is made available through **structp*.

RETURN VALUES

Successful calls to **lwres_string_parse()** and **lwres_addr_parse()** return **LWRES_R_SUCCESS.** Both functions return **LWRES_R_FAILURE** if the buffer is corrupt or **LWRES_R_UNEXPECTEDEND** if the buffer has less space than expected for the components of the encoded string or address.

lwres_getaddrsbyname() returns **LWRES_R_SUCCESS** on success and it returns **LWRES_R_NOTFOUND** if the hostname *name* could not be found.

LWRES_R_SUCCESS is returned by a successful call to **lwres_getnamebyaddr()**.

Both **lwres_getaddrsbyname()** and **lwres_getnamebyaddr()** return **LWRES_R_NOMEMORY** when memory allocation requests fail and **LWRES_R_UNEXPECTEDEND** if the buffers used for sending queries and receiving replies are too small.

SEE ALSO

lwres_buffer(3), **lwres_gabn**(3).

COPYRIGHT

NAME

makecontext, **swapcontext** — manipulate user contexts

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ucontext.h>

void
makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);

int
swapcontext(ucontext_t * restrict oucp, ucontext_t * restrict ucp);
```

DESCRIPTION

The **makecontext**() function modifies the object pointed to by *ucp*, which has been initialized using getcontext(2). When this context is resumed using **swapcontext**() or setcontext(2), program execution continues as if *func* had been called with the arguments specified after *argc* in the call of **makecontext**(). The value of *argc* must be equal to the number of integer arguments following it, and must be equal to the number of integer arguments expected by *func*; otherwise, the behavior is undefined.

Before being modified using **makecontext**(), a stack must be allocated for the context (in the *uc_stack* member), and a context to resume after *func* has returned must be determined (pointed to by the *uc_link* member); otherwise, the behavior is undefined. If *uc_link* is a null pointer, then the context is the main context, and the process will exit with an exit status of 0 upon return.

The **swapcontext**() function saves the current context in the object pointed to by *oucp*, sets the current context to that specified in the object pointed to by *ucp*, and resumes execution. When a context saved by **swapcontext**() is restored using setcontext(2), execution will resume as if the corresponding invocation of **swapcontext**() had just returned (successfully).

RETURN VALUES

The **makecontext**() function returns no value.

On success, **swapcontext**() returns a value of 0, Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

The **swapcontext**() function will fail if:

[EFAULT] The *oucp* argument points to an invalid address.

[EFAULT] The *ucp* argument points to an invalid address.

[EINVAL] The contents of the datum pointed to by *ucp* are invalid.

SEE ALSO

_exit(2), getcontext(2), setcontext(2), ucontext(2)

STANDARDS

The **makecontext**() and **swapcontext**() functions conform to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

The standard does not clearly define the type of integer arguments passed to *func* via makecontext(); portable applications should not rely on the implementation detail that it may be

possible to pass pointer arguments to functions. This may be clarified in a future revision of the standard.

HISTORY

The **makecontext**() and **swapcontext**() functions first appeared in AT&T System V.4 UNIX.

NAME

malloc, **calloc**, **realloc**, **free** — general purpose memory allocation functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

void *
malloc(size_t size);

void *
calloc(size_t number, size_t size);

void *
realloc(void *ptr, size_t size);

void
free(void *ptr);

const char * _malloc_options;
```

DESCRIPTION

The **malloc**() function allocates *size* bytes of uninitialized memory. The allocated space is suitably aligned (after possible pointer coercion) for storage of any type of object.

The **calloc**() function allocates space for *number* objects, each *size* bytes in length. The result is identical to calling **malloc**() with an argument of "number * size", with the exception that the allocated memory is explicitly initialized to zero bytes.

The **realloc**() function changes the size of the previously allocated memory referenced by *ptr* to *size* bytes. The contents of the memory are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the memory is undefined. Upon success, the memory referenced by *ptr* is freed and a pointer to the newly allocated memory is returned. Note that **realloc**() may move the memory allocation, resulting in a different return value than *ptr*. If *ptr* is NULL, the **realloc**() function behaves identically to **malloc**() for the specified size.

The **free**() function causes the allocated memory referenced by *ptr* to be made available for future allocations. If *ptr* is NULL, no action occurs.

TUNING

Once, when the first call is made to one of these memory allocation routines, various flags will be set or reset, which affect the workings of this allocator implementation.

The "name" of the file referenced by the symbolic link named /etc/malloc.conf, the value of the environment variable MALLOC_OPTIONS, and the string pointed to by the global variable *_malloc_options* will be interpreted, in that order, character by character as flags.

Most flags are single letters, where uppercase indicates that the behavior is set, or on, and lowercase means that the behavior is not set, or off.

A All warnings (except for the warning about unknown flags being set) become fatal. The process will call abort(3) in these cases.

H Use madvise(2) when pages within a chunk are no longer in use, but the chunk as a whole cannot yet be deallocated. This is primarily of use when swapping is a real possibility, due to the high overhead of the **madvise**() system call.

J Each byte of new memory allocated by **malloc**(), **realloc**() will be initialized to 0xa5. All memory returned by **free**(), **realloc**() will be initialized to 0x5a. This is intended for debugging and will impact performance negatively.

K Increase/decrease the virtual memory chunk size by a factor of two. The default chunk size is 1 MB. This option can be specified multiple times.

N Increase/decrease the number of arenas by a factor of two. The default number of arenas is four times the number of CPUs, or one if there is a single CPU. This option can be specified multiple times.

P Various statistics are printed at program exit via an `atexit`(3) function. This has the potential to cause deadlock for a multi-threaded process that exits while one or more threads are executing in the memory allocation functions. Therefore, this option should only be used with care; it is primarily intended as a performance tuning aid during application development.

Q Increase/decrease the size of the allocation quantum by a factor of two. The default quantum is the minimum allowed by the architecture (typically 8 or 16 bytes). This option can be specified multiple times.

S Increase/decrease the size of the maximum size class that is a multiple of the quantum by a factor of two. Above this size, power-of-two spacing is used for size classes. The default value is 512 bytes. This option can be specified multiple times.

U Generate "utrace" entries for `ktrace`(1), for all operations. Consult the source for details on this option.

V Attempting to allocate zero bytes will return a `NULL` pointer instead of a valid pointer. (The default behavior is to make a minimal allocation and return a pointer to it.) This option is provided for System V compatibility. This option is incompatible with the "X" option.

X Rather than return failure for any allocation function, display a diagnostic message on `stderr` and cause the program to drop core (using `abort`(3)). This option should be set at compile time by including the following in the source code:

```
_malloc_options = "X";
```

Z Each byte of new memory allocated by **malloc**(), **realloc**() will be initialized to 0. Note that this initialization only happens once for each byte, so **realloc**() call do not zero memory that was previously allocated. This is intended for debugging and will impact performance negatively.

The "J" and "Z" options are intended for testing and debugging. An application which changes its behavior when these options are used is flawed.

IMPLEMENTATION NOTES

This allocator uses multiple arenas in order to reduce lock contention for threaded programs on multi-processor systems. This works well with regard to threading scalability, but incurs some costs. There is a small fixed per-arena overhead, and additionally, arenas manage memory completely independently of each other, which means a small fixed increase in overall memory fragmentation. These overheads are not generally an issue, given the number of arenas normally used. Note that using substantially more arenas than the default is not likely to improve performance, mainly due to reduced cache performance. However, it may make sense to reduce the number of arenas if an application does not make much use of the allocation functions.

Memory is conceptually broken into equal-sized chunks, where the chunk size is a power of two that is greater than the page size. Chunks are always aligned to multiples of the chunk size. This alignment makes it possible to find metadata for user objects very quickly.

User objects are broken into three categories according to size: small, large, and huge. Small objects are no larger than one half of a page. Large objects are smaller than the chunk size. Huge objects are a multiple of the chunk size. Small and large objects are managed by arenas; huge objects are managed separately in a single data structure that is shared by all threads. Huge objects are used by applications infrequently enough that this single data structure is not a scalability issue.

Each chunk that is managed by an arena tracks its contents in a page map as runs of contiguous pages (unused, backing a set of small objects, or backing one large object). The combination of chunk alignment and chunk page maps makes it possible to determine all metadata regarding small and large allocations in constant time.

Small objects are managed in groups by page runs. Each run maintains a bitmap that tracks which regions are in use. Allocation requests that are no more than half the quantum (see the "Q" option) are rounded up to the nearest power of two (typically 2, 4, or 8). Allocation requests that are more than half the quantum, but no more than the maximum quantum-multiple size class (see the "S" option) are rounded up to the nearest multiple of the quantum. Allocation requests that are larger than the maximum quantum-multiple size class, but no larger than one half of a page, are rounded up to the nearest power of two. Allocation requests that are larger than half of a page, but small enough to fit in an arena-managed chunk (see the "K" option), are rounded up to the nearest run size. Allocation requests that are too large to fit in an arena-managed chunk are rounded up to the nearest multiple of the chunk size.

Allocations are packed tightly together, which can be an issue for multi-threaded applications. If you need to assure that allocations do not suffer from cache line sharing, round your allocation requests up to the nearest multiple of the cache line size.

DEBUGGING MALLOC PROBLEMS

The first thing to do is to set the "A" option. This option forces a coredump (if possible) at the first sign of trouble, rather than the normal policy of trying to continue if at all possible.

It is probably also a good idea to recompile the program with suitable options and symbols for debugger support.

If the program starts to give unusual results, coredump or generally behave differently without emitting any of the messages mentioned in the next section, it is likely because it depends on the storage being filled with zero bytes. Try running it with the "Z" option set; if that improves the situation, this diagnosis has been confirmed. If the program still misbehaves, the likely problem is accessing memory outside the allocated area.

Alternatively, if the symptoms are not easy to reproduce, setting the "J" option may help provoke the problem.

In truly difficult cases, the "U" option, if supported by the kernel, can provide a detailed trace of all calls made to these functions.

Unfortunately this implementation does not provide much detail about the problems it detects; the performance impact for storing such information would be prohibitive. There are a number of allocator implementations available on the Internet which focus on detecting and pinpointing problems by trading performance for extra sanity checks and detailed diagnostics.

DIAGNOSTIC MESSAGES

If any of the memory allocation/deallocation functions detect an error or warning condition, a message will be printed to file descriptor `STDERR_FILENO`. Errors will result in the process dumping core. If the "A" option is set, all warnings are treated as errors.

The *_malloc_message* variable allows the programmer to override the function which emits the text strings forming the errors and warnings if for some reason the `stderr` file descriptor is not suitable for this. Please note that doing anything which tries to allocate memory in this function is likely to result in a crash or dead-

lock.

All messages are prefixed by "⟨*progname*⟩: (malloc)".

RETURN VALUES

The **malloc**() and **calloc**() functions return a pointer to the allocated memory if successful; otherwise a NULL pointer is returned and *errno* is set to ENOMEM.

The **realloc**() function returns a pointer, possibly identical to *ptr*, to the allocated memory if successful; otherwise a NULL pointer is returned, and *errno* is set to ENOMEM if the error was the result of an allocation failure. The **realloc**() function always leaves the original buffer intact when an error occurs.

The **free**() function returns no value.

ENVIRONMENT

The following environment variables affect the execution of the allocation functions:

MALLOC_OPTIONS If the environment variable MALLOC_OPTIONS is set, the characters it contains will be interpreted as flags to the allocation functions.

EXAMPLES

To dump core whenever a problem occurs:

```
ln -s 'A' /etc/malloc.conf
```

To specify in the source that a program does no return value checking on calls to these functions:

```
_malloc_options = "X";
```

SEE ALSO

limits(1), madvise(2), mmap(2), sbrk(2), alloca(3), atexit(3), getpagesize(3), memory(3), posix_memalign(3)

STANDARDS

The **malloc**(), **calloc**(), **realloc**() and **free**() functions conform to ISO/IEC 9899:1990 ("ISO C90").

NAME

math − introduction to mathematical library functions

DESCRIPTION

These functions constitute the C math library, *libm.* The link editor searches this library under the "−lm" option. Declarations for these functions may be obtained from the include file <*math.h*>.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*	*Error Bound (ULPs)*
acos	acos.3	inverse trigonometric function	3
acosh	acosh.3	inverse hyperbolic function	3
asin	asin.3	inverse trigonometric function	3
asinh	asinh.3	inverse hyperbolic function	3
atan	atan.3	inverse trigonometric function	1
atanh	atanh.3	inverse hyperbolic function	3
atan2	atan2.3	inverse trigonometric function	2
cbrt	sqrt.3	cube root	1
ceil	ceil.3	integer no less than	0
copysign	ieee.3	copy sign bit	0
cos	cos.3	trigonometric function	1
cosh	cosh.3	hyperbolic function	3
erf	erf.3	error function	???
erfc	erf.3	complementary error function	???
exp	exp.3	exponential	1
expm1	exp.3	exp(x)−1	1
fabs	fabs.3	absolute value	0
finite	ieee.3	test for finity	0
floor	floor.3	integer no greater than	0
fmod	fmod.3	remainder	???
hypot	hypot.3	Euclidean distance	1
ilogb	ieee.3	exponent extraction	0
isinf	isinf.3	test for infinity	0
isnan	isnan.3	test for not-a-number	0
j0	j0.3	Bessel function	???
j1	j0.3	Bessel function	???
jn	j0.3	Bessel function	???
lgamma	lgamma.3	log gamma function	???
log	exp.3	natural logarithm	1
log10	exp.3	logarithm to base 10	3
log1p	exp.3	log(1+x)	1
nan	nan.3	return quiet *NaN*	0
nextafter	ieee.3	next representable number	0
pow	exp.3	exponential x**y	60−500
remainder	ieee.3	remainder	0
rint	rint.3	round to nearest integer	0
scalbn	ieee.3	exponent adjustment	0
sin	sin.3	trigonometric function	1
sinh	sinh.3	hyperbolic function	3
sqrt	sqrt.3	square root	1
tan	tan.3	trigonometric function	3
tanh	tanh.3	hyperbolic function	3
trunc	trunc.3	nearest integral value	3
y0	j0.3	Bessel function	???
y1	j0.3	Bessel function	???
yn	j0.3	Bessel function	???

LIST OF DEFINED VALUES

Name	*Value*	*Description*
M_E	2.7182818284590452354	e
M_LOG2E	1.4426950408889634074	log 2e
M_LOG10E	0.43429448190325182765	log 10e
M_LN2	0.69314718055994530942	log e2
M_LN10	2.30258509299404568402	log e10
M_PI	3.14159265358979323846	pi
M_PI_2	1.57079632679489661923	pi/2
M_PI_4	0.78539816339744830962	pi/4
M_1_PI	0.31830988618379067154	1/pi
M_2_PI	0.63661977236758134308	2/pi
M_2_SQRTPI	1.12837916709551257390	2/sqrt(pi)
M_SQRT2	1.41421356237309504880	sqrt(2)
M_SQRT1_2	0.70710678118654752440	1/sqrt(2)

NOTES

In 4.3 BSD, distributed from the University of California in late 1985, most of the foregoing functions come in two versions, one for the double−precision "D" format in the DEC VAX−11 family of computers, another for double−precision arithmetic conforming to the IEEE Standard 754 for Binary Floating−Point Arithmetic. The two versions behave very similarly, as should be expected from programs more accurate and robust than was the norm when UNIX was born. For instance, the programs are accurate to within the numbers of *ulp*s tabulated above; an *ulp* is one *U*nit in the *L*ast *P*lace. And the programs have been cured of anomalies that afflicted the older math library *libm* in which incidents like the following had been reported:

sqrt(−1.0) = 0.0 and log(−1.0) = −1.7e38.
cos(1.0e−11) > cos(0.0) > 1.0.
pow(x,1.0) ≠ x when x = 2.0, 3.0, 4.0, ..., 9.0.
pow(−1.0,1.0e10) trapped on Integer Overflow.
sqrt(1.0e30) and sqrt(1.0e−30) were very slow.

However the two versions do differ in ways that have to be explained, to which end the following notes are provided.

DEC VAX−11 D_floating−point:

This is the format for which the original math library *libm* was developed, and to which this manual is still principally dedicated. It is *the* double−precision format for the PDP−11 and the earlier VAX−11 machines; VAX−11s after 1983 were provided with an optional "G" format closer to the IEEE double−precision format. The earlier DEC MicroVAXs have no D format, only G double−precision. (Why? Why not?)

Properties of D_floating−point:

Wordsize: 64 bits, 8 bytes. Radix: Binary.

Precision: 56 significant bits, roughly like 17 significant decimals.

If x and x' are consecutive positive D_floating−point numbers (they differ by 1 *ulp*), then 1.3e−17 < 0.5**56 < (x'−x)/x 0.5**55 < 2.8e−17.

Range: Overflow threshold = 2.0**127 = 1.7e38.

Underflow threshold = 0.5**128 = 2.9e−39.

NOTE: THIS RANGE IS COMPARATIVELY NARROW.

Overflow customarily stops computation.

Underflow is customarily flushed quietly to zero.

CAUTION:

It is possible to have x ≠ y and yet x−y = 0 because of underflow. Similarly x > y > 0 cannot prevent either x∗y = 0 or y/x = 0 from happening without warning.

Zero is represented ambiguously.

Although 2**55 different representations of zero are accepted by the hardware, only the obvious representation is ever produced. There is no −0 on a VAX.

∞ is not part of the VAX architecture.

Reserved operands:
of the 2**55 that the hardware recognizes, only one of them is ever produced. Any floating−point operation upon a reserved operand, even a MOVF or MOVD, customarily stops computation, so they are not much used.

Exceptions:
Divisions by zero and operations that overflow are invalid operations that customarily stop computation or, in earlier machines, produce reserved operands that will stop computation.

Rounding:
Every rational operation (+, −, ∗, /) on a VAX (but not necessarily on a PDP−11), if not an over/underflow nor division by zero, is rounded to within half an *ulp*, and when the rounding error is exactly half an *ulp* then rounding is away from 0.

Except for its narrow range, D_floating−point is one of the better computer arithmetics designed in the 1960's. Its properties are reflected fairly faithfully in the elementary functions for a VAX distributed in 4.3 BSD. They over/underflow only if their results have to lie out of range or very nearly so, and then they behave much as any rational arithmetic operation that over/underflowed would behave. Similarly, expressions like log(0) and atanh(1) behave like 1/0; and sqrt(−3) and acos(3) behave like 0/0; they all produce reserved operands and/or stop computation! The situation is described in more detail in manual pages.

This response seems excessively punitive, so it is destined to be replaced at some time in the foreseeable future by a more flexible but still uniform scheme being developed to handle all floating−point arithmetic exceptions neatly.

How do the functions in 4.3 BSD's new *libm* for UNIX compare with their counterparts in DEC's VAX/VMS library? Some of the VMS functions are a little faster, some are a little more accurate, some are more puritanical about exceptions (like pow(0.0,0.0) and atan2(0.0,0.0)), and most occupy much more memory than their counterparts in *libm*. The VMS codes interpolate in large table to achieve speed and accuracy; the *libm* codes use tricky formulas compact enough that all of them may some day fit into a ROM.

More important, DEC regards the VMS codes as proprietary and guards them zealously against unauthorized use. But the *libm* codes in 4.3 BSD are intended for the public domain; they may be copied freely provided their provenance is always acknowledged, and provided users assist the authors in their researches by reporting experience with the codes. Therefore no user of UNIX on a machine whose arithmetic resembles VAX D_floating−point need use anything worse than the new *libm*.

IEEE STANDARD 754 Floating−Point Arithmetic:

This standard is on its way to becoming more widely adopted than any other design for computer arithmetic. VLSI chips that conform to some version of that standard have been produced by a host of manufacturers, among them ...

Intel i8087, i80287	National Semiconductor 32081
Motorola 68881	Weitek WTL-1032, ... , -1165
Zilog Z8070	Western Electric (AT&T) WE32106.

Other implementations range from software, done thoroughly in the Apple Macintosh, through VLSI in the Hewlett−Packard 9000 series, to the ELXSI 6400 running ECL at 3 Megaflops. Several other companies have adopted the formats of IEEE 754 without, alas, adhering to the standard's way of handling rounding and exceptions like over/underflow. The DEC VAX G_floating−point format is very similar to the IEEE 754 Double format, so similar that the C programs for the IEEE versions of most of the elementary functions listed above could easily be converted to run on a MicroVAX, though nobody has volunteered to do that yet.

The codes in 4.3 BSD's *libm* for machines that conform to IEEE 754 are intended primarily for the National Semi. 32081 and WTL 1164/65. To use these codes with the Intel or Zilog chips, or with the Apple Macintosh or ELXSI 6400, is to forego the use of better codes provided (perhaps freely) by those companies and designed by some of the authors of the codes above. Except for *atan*, *cbrt*, *erf*, *erfc*, *hypot*, *j0−jn*, *lgamma*, *pow* and *y0−yn*, the Motorola 68881 has all the functions in *libm* on chip, and faster and more accurate; it, Apple, the i8087, Z8070 and WE32106 all use 64 significant bits. The main virtue of 4.3

BSD's *libm* codes is that they are intended for the public domain; they may be copied freely provided their provenance is always acknowledged, and provided users assist the authors in their researches by reporting experience with the codes. Therefore no user of UNIX on a machine that conforms to IEEE 754 need use anything worse than the new *libm*.

Properties of IEEE 754 Double−Precision:

Wordsize: 64 bits, 8 bytes. Radix: Binary.

Precision: 53 significant bits, roughly like 16 significant decimals.

If x and x' are consecutive positive Double−Precision numbers (they differ by 1 *ulp*), then 1.1e−16 < 0.5**53 < (x'−x)/x 0.5**52 < 2.3e−16.

Range: Overflow threshold = 2.0**1024 = 1.8e308

Underflow threshold = 0.5**1022 = 2.2e−308

Overflow goes by default to a signed ∞.

Underflow is *Gradual*, rounding to the nearest integer multiple of 0.5**1074 = 4.9e−324.

Zero is represented ambiguously as +0 or −0.

Its sign transforms correctly through multiplication or division, and is preserved by addition of zeros with like signs; but x−x yields +0 for every finite x. The only operations that reveal zero's sign are division by zero and copysign(x,±0). In particular, comparison (x > y, x y, etc.) cannot be affected by the sign of zero; but if finite x = y then ∞ = 1/(x−y) ≠ −1/(y−x) = −∞.

∞ is signed.

it persists when added to itself or to any finite number. Its sign transforms correctly through multiplication and division, and (finite)/±∞ = ±0 (nonzero)/0 = ±∞. But ∞−∞, ∞∗0 and ∞/∞ are, like 0/0 and sqrt(−3), invalid operations that produce *NaN*. ...

Reserved operands:

there are 2**53−2 of them, all called *NaN* (*N*ot *a N*umber). Some, called Signaling *NaN*s, trap any floating−point operation performed upon them; they are used to mark missing or uninitialized values, or nonexistent elements of arrays. The rest are Quiet *NaN*s; they are the default results of Invalid Operations, and propagate through subsequent arithmetic operations. If x ≠ x then x is *NaN*; every other predicate (x > y, x = y, x < y, ...) is FALSE if *NaN* is involved.

NOTE: Trichotomy is violated by *NaN*.

Besides being FALSE, predicates that entail ordered comparison, rather than mere (in)equality, signal Invalid Operation when *NaN* is involved.

Rounding:

Every algebraic operation (+, −, ∗, /, √) is rounded by default to within half an *ulp*, and when the rounding error is exactly half an *ulp* then the rounded value's least significant bit is zero. This kind of rounding is usually the best kind, sometimes provably so; for instance, for every x = 1.0, 2.0, 3.0, 4.0, ..., 2.0**52, we find (x/3.0)∗3.0 == x and (x/10.0)∗10.0 == x and ... despite that both the quotients and the products have been rounded. Only rounding like IEEE 754 can do that. But no single kind of rounding can be proved best for every circumstance, so IEEE 754 provides rounding towards zero or towards +∞ or towards −∞ at the programmer's option. And the same kinds of rounding are specified for Binary−Decimal Conversions, at least for magnitudes between roughly 1.0e−10 and 1.0e37.

Exceptions:

IEEE 754 recognizes five kinds of floating−point exceptions, listed below in declining order of probable importance.

Exception	Default Result
Invalid Operation	*NaN*, or FALSE
Overflow	±∞
Divide by Zero	±∞
Underflow	Gradual Underflow
Inexact	Rounded value

NOTE: An Exception is not an Error unless handled badly. What makes a class of exceptions exceptional is that no single default response can be satisfactory in every instance. On the other hand, if a default response will serve most instances satisfactorily, the unsatisfactory instances cannot justify aborting computation every time the exception occurs.

For each kind of floating−point exception, IEEE 754 provides a Flag that is raised each time its exception is signaled, and stays raised until the program resets it. Programs may also test, save and restore a flag. Thus, IEEE 754 provides three ways by which programs may cope with exceptions for which the default result might be unsatisfactory:

1) Test for a condition that might cause an exception later, and branch to avoid the exception.

2) Test a flag to see whether an exception has occurred since the program last reset its flag.

3) Test a result to see whether it is a value that only an exception could have produced.
CAUTION: The only reliable ways to discover whether Underflow has occurred are to test whether products or quotients lie closer to zero than the underflow threshold, or to test the Underflow flag. (Sums and differences cannot underflow in IEEE 754; if $x \neq y$ then $x-y$ is correct to full precision and certainly nonzero regardless of how tiny it may be.) Products and quotients that underflow gradually can lose accuracy gradually without vanishing, so comparing them with zero (as one might on a VAX) will not reveal the loss. Fortunately, if a gradually underflowed value is destined to be added to something bigger than the underflow threshold, as is almost always the case, digits lost to gradual underflow will not be missed because they would have been rounded off anyway. So gradual underflows are usually *provably* ignorable. The same cannot be said of underflows flushed to 0.

At the option of an implementor conforming to IEEE 754, other ways to cope with exceptions may be provided:

4) ABORT. This mechanism classifies an exception in advance as an incident to be handled by means traditionally associated with error−handling statements like "ON ERROR GO TO ...". Different languages offer different forms of this statement, but most share the following characteristics:

— No means is provided to substitute a value for the offending operation's result and resume computation from what may be the middle of an expression. An exceptional result is abandoned.

— In a subprogram that lacks an error−handling statement, an exception causes the subprogram to abort within whatever program called it, and so on back up the chain of calling subprograms until an error−handling statement is encountered or the whole task is aborted and memory is dumped.

5) STOP. This mechanism, requiring an interactive debugging environment, is more for the programmer than the program. It classifies an exception in advance as a symptom of a programmer's error; the exception suspends execution as near as it can to the offending operation so that the programmer can look around to see how it happened. Quite often the first several exceptions turn out to be quite unexceptionable, so the programmer ought ideally to be able to resume execution after each one as if execution had not been stopped.

6) ... Other ways lie beyond the scope of this document.

The crucial problem for exception handling is the problem of Scope, and the problem's solution is understood, but not enough manpower was available to implement it fully in time to be distributed in 4.3 BSD's *libm*. Ideally, each elementary function should act as if it were indivisible, or atomic, in the sense that ...

i) No exception should be signaled that is not deserved by the data supplied to that function.

ii) Any exception signaled should be identified with that function rather than with one of its subroutines.

iii) The internal behavior of an atomic function should not be disrupted when a calling program changes from one to another of the five or so ways of handling exceptions listed above, although the definition

of the function may be correlated intentionally with exception handling.

Ideally, every programmer should be able *conveniently* to turn a debugged subprogram into one that appears atomic to its users. But simulating all three characteristics of an atomic function is still a tedious affair, entailing hosts of tests and saves−restores; work is under way to ameliorate the inconvenience.

Meanwhile, the functions in *libm* are only approximately atomic. They signal no inappropriate exception except possibly ...

Over/Underflow
when a result, if properly computed, might have lain barely within range, and

Inexact in *cbrt*, *hypot*, *log10* and *pow*
when it happens to be exact, thanks to fortuitous cancellation of errors.

Otherwise, ...

Invalid Operation is signaled only when
any result but *NaN* would probably be misleading.

Overflow is signaled only when
the exact result would be finite but beyond the overflow threshold.

Divide−by−Zero is signaled only when
a function takes exactly infinite values at finite operands.

Underflow is signaled only when
the exact result would be nonzero but tinier than the underflow threshold.

Inexact is signaled only when
greater range or precision would be needed to represent the exact result.

SEE ALSO

An explanation of IEEE 754 and its proposed extension p854 was published in the IEEE magazine MICRO in August 1984 under the title "A Proposed Radix− and Word−length−independent Standard for Floating−point Arithmetic" by W. J. Cody et al. The manuals for Pascal, C and BASIC on the Apple Macintosh document the features of IEEE 754 pretty well. Articles in the IEEE magazine COMPUTER vol. 14 no. 3 (Mar. 1981), and in the ACM SIGNUM Newsletter Special Issue of Oct. 1979, may be helpful although they pertain to superseded drafts of the standard.

BUGS

When signals are appropriate, they are emitted by certain operations within the codes, so a subroutine−trace may be needed to identify the function with its signal in case method 5) above is in use. And the codes all take the IEEE 754 defaults for granted; this means that a decision to trap all divisions by zero could disrupt a code that would otherwise get correct results despite division by zero.

NAME

mblen — get number of bytes in a multibyte character

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
mblen(const char *s, size_t n);
```

DESCRIPTION

The **mblen**() function usually determines the number of bytes in a multibyte character pointed to by *s* and returns it. This function shall only examine max n bytes of the array beginning from *s*.

In state-dependent encodings, *s* may point the special sequence bytes to change the shift-state. Although such sequence bytes corresponds to no individual wide-character code, the **mblen**() changes the own state by them and treats them as if they are a part of the subsequent multibyte character.

Unlike mbrlen(3), the first *n* bytes pointed to by *s* need to form an entire multibyte character. Otherwise, this function causes an error.

mblen() is equivalent to the following call, except the internal state of the mbtowc(3) function is not affected:

```
mbtowc(NULL, s, n);
```

Calling any other functions in Standard C Library (libc, −lc) never changes the internal state of **mblen**(), except for calling setlocale(3) with the LC_CTYPE category changed to that of the current locale. Such setlocale(3) calls cause the internal state of this function to be indeterminate.

The behaviour of **mblen**() is affected by the LC_CTYPE category of the current locale.

These are the special cases:

s == NULL — **mblen**() initializes its own internal state to an initial state, and determines whether the current encoding is state-dependent. This function returns 0 if the encoding is state-independent, otherwise non-zero.

n == 0 — In this case, the first *n* bytes of the array pointed to by *s* never form a complete character. Thus, **mblen**() always fails.

RETURN VALUES

Normally, **mblen**() returns:

0 — *s* points to a nul byte ('\0').

positive — The value returned is a number of bytes for the valid multibyte character pointed to by *s*. There are no cases that this value is greater than *n* or the value of the MB_CUR_MAX macro.

-1 — *s* points to an invalid or incomplete multibyte character. The **mblen**() also sets *errno* to indicate the error.

When *s* is equal to NULL, the **mblen**() returns:

0 — The current encoding is state-independent.

non-zero The current encoding is state-dependent.

ERRORS

mblen() may cause an error in the following case:

[EILSEQ] *s* points to an invalid or incomplete multibyte character.

SEE ALSO

mbrlen(3), mbtowc(3), setlocale(3)

STANDARDS

The **mblen**() function conforms to ANSI X3.159-1989 ("ANSI C89").

NAME

mbrlen — get number of bytes in a multibyte character (restartable)

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

size_t
mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);
```

DESCRIPTION

The **mbrlen**() function usually determines the number of bytes in a multibyte character pointed to by *s* and returns it. This function shall only examine max n bytes of the array beginning from *s*.

mbrlen() is equivalent to the following call (except *ps* is evaluated only once):

```
mbrtowc(NULL, s, n, (ps != NULL) ? ps : &internal);
```

Here, *internal* is an internal state object.

In state-dependent encodings, *s* may point to the special sequence bytes to change the shift-state. Although such sequence bytes corresponds to no individual wide-character code, these affect the conversion state object pointed to by *ps*, and the **mbrlen**() treats the special sequence bytes as if these are a part of the subsequent multibyte character.

Unlike mblen(3), **mbrlen**() may accept the byte sequence when it is not a complete character but possibly contains part of a valid character. In this case, this function will accept all such bytes and save them into the conversion state object pointed to by *ps*. They will be used on subsequent calls of this function to restart the conversion suspended.

The behaviour of **mbrlen**() is affected by the LC_CTYPE category of the current locale.

These are the special cases:

s == NULL
: **mbrlen**() sets the conversion state object pointed to by *ps* to an initial state and always returns 0. Unlike mblen(3), the value returned does not indicate whether the current encoding of the locale is state-dependent.

  In this case, **mbrlen**() ignores *n*.

n == 0
: In this case, the first *n* bytes of the array pointed to by *s* never form a complete character. Thus, **mbrlen**() always returns (size_t)-2.

ps == NULL
: **mbrlen**() uses its own internal state object to keep the conversion state, instead of *ps* mentioned in this manual page.

  Calling any other functions in Standard C Library (libc, −lc) never changes the internal state of **mbrlen**(), except for calling setlocale(3) with a changing LC_CTYPE category of the current locale. Such setlocale(3) calls cause the internal state of this function to be indeterminate. This internal state is initialized at startup time of the program.

RETURN VALUES

The **mbrlen**() returns:

0
: *s* points to a nul byte ('\0').

positive — The value returned is a number of bytes for the valid multibyte character pointed to by *s*. There are no cases that this value is greater than *n* or the value of the MB_CUR_MAX macro.

(size_t)-2 — *s* points to the byte sequence which possibly contains part of a valid multibyte character, but which is incomplete. When *n* is at least MB_CUR_MAX, this case can only occur if the array pointed to by *s* contains a redundant shift sequence.

(size_t)-1 — *s* points to an illegal byte sequence which does not form a valid multibyte character. In this case, **mbrtowc**() sets *errno* to indicate the error.

ERRORS

mbrlen() may cause an error in the following case:

[EILSEQ] — *s* points to an invalid multibyte character.

[EINVAL] — *ps* points to an invalid or uninitialized mbstate_t object.

SEE ALSO

mblen(3), mbrtowc(3), setlocale(3)

STANDARDS

The **mbrlen**() function conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”). The restrict qualifier is added at ISO/IEC 9899:1999 (“ISO C99”).

NAME

mbrtowc — converts a multibyte character to a wide character (restartable)

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

size_t
mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n,
    mbstate_t * restrict ps);
```

DESCRIPTION

The **mbrtowc**() usually converts the multibyte character pointed to by *s* to a wide character, and stores the wide character to the wchar_t object pointed to by *pwc* if *pwc* is non-NULL and *s* points to a valid character. The conversion happens in accordance with, and changes the conversion state described in the mbstate_t object pointed to by *ps*. This function may examine at most *n* bytes of the array beginning from *s*.

If *s* points to a valid character and the character corresponds to a nul wide character, then the **mbrtowc**() places the mbstate_t object pointed to by *ps* to an initial conversion state.

Unlike mbtowc(3), the **mbrtowc**() may accept the byte sequence pointed to by *s* not forming a complete multibyte character but which may be part of a valid character. In this case, this function will accept all such bytes and save them into the conversion state object pointed to by *ps*. They will be used at subsequent calls of this function to restart the conversion suspended.

The behaviour of **mbrtowc**() is affected by the LC_CTYPE category of the current locale.

These are the special cases:

s == NULL
: **mbrtowc**() sets the conversion state object pointed to by *ps* to an initial state and always returns 0. Unlike mbtowc(3), the value returned does not indicate whether the current encoding of the locale is state-dependent.

  In this case, **mbrtowc**() ignores *pwc* and *n*, and is equivalent to the following call:

```
mbrtowc(NULL, "", 1, ps);
```

pwc == NULL
: The conversion from a multibyte character to a wide character has taken place and the conversion state may be affected, but the resulting wide character is discarded.

ps == NULL
: **mbrtowc**() uses its own internal state object to keep the conversion state, instead of *ps* mentioned in this manual page.

  Calling any other functions in Standard C Library (libc, −lc) never changes the internal state of **mbrtowc**(), which is initialized at startup time of the program.

RETURN VALUES

In the usual cases, **mbrtowc**() returns:

0
: The next bytes pointed to by *s* form a nul character.

positive
: If *s* points to a valid character, **mbrtowc**() returns the number of bytes in the character.

(size_t)-2
: *s* points to a byte sequence which possibly contains part of a valid multibyte character, but which is incomplete. When *n* is at least MB_CUR_MAX, this case can only occur if the array pointed to by *s* contains a redundant shift sequence.

(size_t)-1 *s* points to an illegal byte sequence which does not form a valid multibyte character. In this case, **mbrtowc**() sets *errno* to indicate the error.

ERRORS

mbrtowc() may cause an error in the following case:

[EILSEQ] *s* points to an invalid or incomplete multibyte character.

[EINVAL] *ps* points to an invalid or uninitialized mbstate_t object.

SEE ALSO

mbrlen(3), mbtowc(3), setlocale(3)

STANDARDS

The **mbrtowc**() function conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”). The restrict qualifier is added at ISO/IEC 9899:1999 (“ISO C99”).

NAME

mbsinit — determines whether the state object is in the initial state

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

int
mbsinit(const mbstate_t *ps);
```

DESCRIPTION

The **mbsinit**() determines whether the state object pointed to by *ps* is the initial conversion state, or not.

ps may be a null pointer. In this case, **mbsinit**() will always return non-zero.

RETURN VALUES

mbsinit() returns:

0	The current state is not the initial state.
non-zero	The current state is the initial state or *ps* is a null pointer.

ERRORS

No errors are defined.

STANDARDS

The **mbsinit**() conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”).

NAME

mbsrtowcs — converts a multibyte character string to a wide character string (restartable)

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

size_t
mbsrtowcs(wchar_t * restrict pwcs, const char ** restrict s, size_t n,
    mbstate_t * restrict ps);
```

DESCRIPTION

The **mbsrtowcs**() converts the multibyte character string indirectly pointed to by *s* to the corresponding wide character string, and stores it in the array pointed to by *pwcs*. The conversion stops due to the following reasons:

- The conversion reaches a nul byte. In this case, the nul byte is also converted.
- The **mbsrtowcs**() has already stored *n* wide characters.
- The conversion encounters an invalid character.

Each character will be converted as if mbrtowc(3) is continuously called.

After conversion, if *pwcs* is not a null pointer, the pointer object pointed to by *s* is a null pointer (if the conversion is stopped due to reaching a nul byte) or the first byte of the character just after the last character converted.

If *pwcs* is not a null pointer and the conversion is stopped due to reaching a nul byte, the **mbsrtowcs**() places the state object pointed to by *ps* to an initial state after the conversion has taken place.

The behaviour of **mbsrtowcs**() is affected by the LC_CTYPE category of the current locale.

These are the special cases:

s == NULL || *s == NULL
: Undefined (may cause the program to crash).

pwcs == NULL
: The conversion has taken place, but the resulting wide character string was discarded. In this case, the pointer object pointed to by *s* is not modified and *n* is ignored.

ps == NULL
: The **mbsrtowcs**() uses its own internal state object to keep the conversion state, instead of *ps* mentioned in this manual page.

  Calling any other functions in Standard C Library (libc, −lc) never changes the internal state of **mbsrtowcs**(), which is initialized at startup time of the program.

RETURN VALUES

mbsrtowcs() returns:

0 or positive
: The value returned is the number of elements stored in the array pointed to by *pwcs*, except for a terminating nul wide character (if any). If *pwcs* is not NULL and the value returned is equal to *n*, the wide character string pointed to by *pwcs* is not nul-terminated. If *pwcs* is a null pointer, the value returned is the number of elements to contain the whole string converted, except for a terminating nul wide character.

(size_t)-1 The array indirectly pointed to by *s* contains a byte sequence forming invalid character. In this case, **mbsrtowcs**() sets *errno* to indicate the error.

ERRORS

mbsrtowcs() may cause an error in the following case:

[EILSEQ] The pointer pointed to by *s* points to an invalid or incomplete multibyte character.

[EINVAL] *ps* points to an invalid or uninitialized mbstate_t object.

SEE ALSO

mbrtowc(3), mbstowcs(3), setlocale(3)

STANDARDS

The **mbsrtowcs**() function conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”). The restrict qualifier is added at ISO/IEC 9899:1999 (“ISO C99”).

NAME

mbstowcs — converts a multibyte character string to a wide character string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

size_t
mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);
```

DESCRIPTION

mbstowcs() converts a nul-terminated multibyte character string pointed to by *s* to the corresponding wide character string and stores it in the array pointed to by *pwcs*. This function may modify the first at most *n* elements of the array pointed to by *pwcs*. Each character will be converted as if mbtowc(3) is continuously called, except the internal state of mbtowc(3) will not be affected.

For state-dependent encoding, **mbstowcs**() implies the multibyte character string pointed to by *s* always begins with an initial state.

These are the special cases:

pwcs == NULL **mbstowcs**() returns the number of elements to store the whole wide character string corresponding to the multibyte character string pointed to by *s*. In this case, *n* is ignored.

s == NULL Undefined (may cause the program to crash).

RETURN VALUES

mbstowcs() returns:

0 or positive Number of elements stored in the array pointed to by *pwcs*. There are no cases that the value returned is greater than *n* (unless *pwcs* is a null pointer) or the value of the MB_CUR_MAX macro. If the return value is equal to *n*, the string pointed to by *pwcs* will not be nul-terminated.

(size_t)-1 *s* points to a string containing an invalid or incomplete multibyte character. The **mbstowcs**() also sets *errno* to indicate the error.

ERRORS

mbstowcs() may cause an error in the following case:

[EILSEQ] *s* points to a string containing an invalid or incomplete multibyte character.

SEE ALSO

mbtowc(3), setlocale(3)

STANDARDS

The **mbstowcs**() function conforms to ANSI X3.159-1989 (“ANSI C89”). The restrict qualifier is added at ISO/IEC 9899:1999 (“ISO C99”).

NAME

mbtowc — converts a multibyte character to a wide character

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
mbtowc(wchar_t * restrict pwc, const char * restrict s, size_t n);
```

DESCRIPTION

mbtowc() usually converts the multibyte character pointed to by *s* to a wide character, and stores it in the wchar_t object pointed to by *pwc* if *pwc* is non-NULL and *s* points to a valid character. This function may inspect at most n bytes of the array beginning from *s*.

In state-dependent encodings, *s* may point to the special sequence bytes to change the shift-state. Although such sequence bytes correspond to no individual wide-character code, **mbtowc**() changes its own state by the sequence bytes and treats them as if they are a part of the subsequence multibyte character.

Unlike mbrtowc(3), the first *n* bytes pointed to by *s* need to form an entire multibyte character. Otherwise, this function causes an error.

Calling any other functions in Standard C Library (libc, −lc) never changes the internal state of **mbtowc**(), except for calling setlocale(3) with changing the LC_CTYPE category of the current locale. Such setlocale(3) call causes the internal state of this function to be indeterminate.

The behaviour of **mbtowc**() is affected by the LC_CTYPE category of the current locale.

There are special cases:

s == NULL — **mbtowc**() initializes its own internal state to an initial state, and determines whether the current encoding is state-dependent. This function returns 0 if the encoding is state-independent, otherwise non-zero. In this case, *pwc* is completely ignored.

pwc == NULL — **mbtowc**() executes the conversion as if *pwc* is non-NULL, but a result of the conversion is discarded.

n == 0 — In this case, the first *n* bytes of the array pointed to by *s* never form a complete character. Thus, the **mbtowc**() always fails.

RETURN VALUES

Normally, the **mbtowc**() returns:

0 — *s* points to a nul byte ('\0').

positive — Number of bytes for the valid multibyte character pointed to by *s*. There are no cases that the value returned is greater than the value of the MB_CUR_MAX macro.

-1 — *s* points to an invalid or an incomplete multibyte character. The **mbtowc**() also sets *errno* to indicate the error.

When *s* is equal to NULL, **mbtowc**() returns:

0 — The current encoding is state-independent.

non-zero The current encoding is state-dependent.

ERRORS

mbtowc() may cause an error in the following case:

[EILSEQ] *s* points to an invalid or incomplete multibyte character.

SEE ALSO

mblen(3), mbrtowc(3), setlocale(3)

STANDARDS

The **mbtowc**() function conforms to ANSI X3.159-1989 (“ANSI C89”). The restrict qualifier is added at ISO/IEC 9899:1999 (“ISO C99”).

NAME

MD2Init, **MD2Update**, **MD2Final**, **MD2End**, **MD2File**, **MD2Data** — calculate the RSA Data Security, Inc., “MD2” message digest

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <mdX.h>

void
MD2Init(MD2_CTX *context);

void
MD2Update(MD2_CTX *context, const unsigned char *data, unsigned int len);

void
MD2Final(unsigned char digest[16], MD2_CTX *context);

char *
MD2End(MD2_CTX *context, char *buf);

char *
MD2File(const char *filename, char *buf);

char *
MD2Data(const unsigned char *data, unsigned int len, char *buf);
```

DESCRIPTION

The MD2 functions calculate a 128-bit cryptographic checksum (digest) for any number of input bytes. A cryptographic checksum is a one-way hash-function, that is, you cannot find (except by exhaustive search) the input corresponding to a particular output. This net result is a ‘‘fingerprint’’ of the input-data, which doesn’t disclose the actual input.

The MD2 routines should not be used for any security-related purpose.

The **MD2Init**(), **MD2Update**(), and **MD2Final**() functions are the core functions. Allocate an MD2_CTX, initialize it with **MD2Init**(), run over the data with **MD2Update**(), and finally extract the result using **MD2Final**().

MD2End() is a wrapper for **MD2Final**() which converts the return value to a 33-character (including the terminating ’\0’) ASCII string which represents the 128 bits in hexadecimal.

MD2File() calculates the digest of a file, and uses **MD2End**() to return the result. If the file cannot be opened, a null pointer is returned. **MD2Data**() calculates the digest of a chunk of data in memory, and uses **MD2End**() to return the result.

When using **MD2End**(), **MD2File**(), or **MD2Data**(), the *buf* argument can be a null pointer, in which case the returned string is allocated with `malloc`(3) and subsequently must be explicitly deallocated using `free`(3) after use. If the *buf* argument is non-null it must point to at least 33 characters of buffer space.

SEE ALSO

`md2`(3),

B. Kaliski, *The MD2 Message-Digest Algorithm*, RFC 1319.

RSA Laboratories, *Frequently Asked Questions About today's Cryptography*.

HISTORY

These functions appeared in NetBSD 1.3.

AUTHORS

The original MD2 routines were developed by RSA Data Security, Inc., and published in the above references. This code is a public domain implementation by Andrew Brown.

BUGS

No method is known to exist which finds two files having the same hash value, nor to find a file with a specific hash value. There is on the other hand no guarantee that such a method doesn't exist.

NAME

MDXInit, **MDXUpdate**, **MDXFinal**, **MDXEnd**, **MDXFile**, **MDXData** — calculate the RSA Data Security, Inc., “MDX” message digest

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <mdX.h>

void
MDXInit(MDX_CTX *context);

void
MDXUpdate(MDX_CTX *context, const unsigned char *data, unsigned int len);

void
MDXFinal(unsigned char digest[16], MDX_CTX *context);

char *
MDXEnd(MDX_CTX *context, char *buf);

char *
MDXFile(const char *filename, char *buf);

char *
MDXData(const unsigned char *data, unsigned int len, char *buf);
```

DESCRIPTION

The MDX functions calculate a 128-bit cryptographic checksum (digest) for any number of input bytes. A cryptographic checksum is a one-way hash-function, that is, you cannot find (except by exhaustive search) the input corresponding to a particular output. This net result is a ‘‘fingerprint’’ of the input-data, which doesn’t disclose the actual input.

MD2 is the slowest, MD4 is the fastest and MD5 is somewhere in the middle. MD2 can only be used for Privacy-Enhanced Mail. MD4 has been criticized for being too weak, so MD5 was developed in response as ‘‘MD4 with safety-belts’’. When in doubt, use MD5.

The **MDXInit**(), **MDXUpdate**(), and **MDXFinal**() functions are the core functions. Allocate an MDX_CTX, initialize it with **MDXInit**(), run over the data with **MDXUpdate**(), and finally extract the result using **MDXFinal**().

MDXEnd() is a wrapper for **MDXFinal**() which converts the return value to a 33-character (including the terminating ’\0’) ASCII string which represents the 128 bits in hexadecimal.

MDXFile() calculates the digest of a file, and uses **MDXEnd**() to return the result. If the file cannot be opened, a null pointer is returned. **MDXData**() calculates the digest of a chunk of data in memory, and uses **MDXEnd**() to return the result.

When using **MDXEnd**(), **MDXFile**(), or **MDXData**(), the *buf* argument can be a null pointer, in which case the returned string is allocated with malloc(3) and subsequently must be explicitly deallocated using free(3) after use. If the *buf* argument is non-null it must point to at least 33 characters of buffer space.

SEE ALSO

md2(3), md4(3), md5(3)

B. Kaliski, *The MD2 Message-Digest Algorithm*, RFC 1319.

R. Rivest, *The MD4 Message-Digest Algorithm*, RFC 1186.

R. Rivest, *The MD5 Message-Digest Algorithm*, RFC 1321.

RSA Laboratories, *Frequently Asked Questions About today's Cryptography*.

HISTORY

These functions appeared in NetBSD 1.3.

AUTHORS

The original MDX routines were developed by RSA Data Security, Inc., and published in the above references. This code is derived directly from these implementations by Poul-Henning Kamp ⟨phk@login.dkuug.dk⟩

Phk ristede runen.

BUGS

No method is known to exist which finds two files having the same hash value, nor to find a file with a specific hash value. There is on the other hand no guarantee that such a method doesn't exist.

COPYRIGHT

NAME

membar_ops, **membar_enter**, **membar_exit**, **membar_producer**, **membar_consumer**, **membar_sync** — memory access barrier operations

SYNOPSIS

```
#include <sys/atomic.h>

void
membar_enter(void);

void
membar_exit(void);

void
membar_producer(void);

void
membar_consumer(void);

void
membar_sync(void);
```

DESCRIPTION

The **membar_ops** family of functions provide memory access barrier operations necessary for synchronization in multiprocessor execution environments that have relaxed load and store order.

membar_enter()

Any store preceeding **membar_enter**() will reach global visibility before all loads and stores following it.

membar_enter() is typically used in code that implements locking primitives to ensure that a lock protects its data.

membar_exit()

All loads and stores preceding **membar_exit**() will reach global visibility before any store that follows it.

membar_exit() is typically used in code that implements locking primitives to ensure that a lock protects its data.

membar_producer()

All stores preceding the memory barrier will reach global visibility before any stores after the memory barrier reach global visibility.

membar_consumer()

All loads preceding the memory barrier will complete before any loads after the memory barrier complete.

membar_sync()

All loads and stores preceding the memory barrier will complete and reach global visibility before any loads and stores after the memory barrier complete and reach global visibility.

SEE ALSO

atomic_ops(3)

HISTORY

The **membar_ops** functions first appeared in NetBSD 5.0.

NAME

memccpy — copy string until character found

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

void *
memccpy(void *dst, const void *src, int c, size_t len);
```

DESCRIPTION

The **memccpy**() function copies bytes from string *src* to string *dst*. If the character *c* (as converted to an unsigned char) occurs in the string *src*, the copy stops and a pointer to the byte after the copy of *c* in the string *dst* is returned. Otherwise, *len* bytes are copied, and a null pointer is returned.

SEE ALSO

bcopy(3), memcpy(3), memmove(3), strcpy(3)

HISTORY

The **memccpy**() function first appeared in 4.4BSD.

NAME

memchr — locate byte in byte string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

void *
memchr(const void *b, int c, size_t len);
```

DESCRIPTION

The **memchr**() function locates the first occurrence of *c* (converted to an unsigned char) in string *b*.

RETURN VALUES

The **memchr**() function returns a pointer to the byte located, or NULL if no such byte exists within *len* bytes.

SEE ALSO

index(3), rindex(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **memchr**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

memcmp — compare byte string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

int
memcmp(const void *b1, const void *b2, size_t len);
```

DESCRIPTION

The **memcmp**() function compares byte string *b1* against byte string *b2*. Both strings are assumed to be *len* bytes long.

RETURN VALUES

The **memcmp**() function returns zero if the two strings are identical, otherwise returns the difference between the first two differing bytes (treated as unsigned char values, so that '`\200`' is greater than '`\0`', for example). Zero-length strings are always identical.

SEE ALSO

`bcmp`(3), `strcasecmp`(3), `strcmp`(3), `strcoll`(3), `strxfrm`(3)

STANDARDS

The **memcmp**() function conforms to ANSI X3.159-1989 ("ANSI C89").

NAME

memcpy — copy byte string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

void *
memcpy(void * restrict dst, const void * restrict src, size_t len);
```

DESCRIPTION

The **memcpy**() function copies *len* bytes from string *src* to string *dst*. The arguments must not overlap -- behavior if the arguments overlap is undefined. To copy byte strings that overlap, use memmove(3).

RETURN VALUES

The **memcpy**() function returns the original value of *dst*.

SEE ALSO

bcopy(3), memccpy(3), memmove(3), strcpy(3)

STANDARDS

The **memcpy**() function conforms to ISO/IEC 9899:1999 (“ISO C99”).

NAME

memmem — locate substring in byte string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

void *
memmem(const void *block, size_t blen, const void *pat, size_t plen);
```

DESCRIPTION

The **memmem**() function locates the first occurrence of the binary string *pat* of size *plen* bytes in the byte string *block* of size *blen* bytes.

RETURN VALUES

The **memmem**() function returns a pointer to the substring located, or NULL if no such substring exists within *block*.

If *plen* is zero, *block* is returned, i.e. a zero length *pat* is deemed to match the start of the string, as with strstr(3).

SEE ALSO

bm(3), memchr(3), strchr(3), strstr(3)

STANDARDS

The **memmem**() function is not currently standardized. However, it is meant to be API compatible with functions in FreeBSD and Linux.

HISTORY

memmem() first appeared in the Free Software Foundation's glibc library.

NAME

memmove — copy byte string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

void *
memmove(void *dst, const void *src, size_t len);
```

DESCRIPTION

The **memmove**() function copies *len* bytes from string *src* to string *dst*. The two strings may overlap; the copy is always done in a non-destructive manner.

RETURN VALUES

The **memmove**() function returns the original value of *dst*.

SEE ALSO

bcopy(3), memccpy(3), memcpy(3), strcpy(3)

STANDARDS

The **memmove**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

malloc, **free**, **realloc**, **calloc**, **alloca** — general memory allocation operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

void *
malloc(size_t size);

void
free(void *ptr);

void *
realloc(void *ptr, size_t size);

void *
calloc(size_t nelem, size_t elsize);

void *
alloca(size_t size);
```

DESCRIPTION

These functions allocate and free memory for the calling process. They are described in the individual manual pages.

SEE ALSO

alloca(3), calloc(3), free(3), malloc(3), realloc(3)

STANDARDS

These functions, with the exception of **alloca**() conform to ANSI X3.159-1989 (“ANSI C89”).

NAME

memset — write a byte to byte string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

void *
memset(void *b, int c, size_t len);
```

DESCRIPTION

The **memset**() function writes *len* bytes of value *c* (converted to an unsigned char) to the string *b*.

RETURN VALUES

The **memset**() function returns the original value of *b*.

SEE ALSO

bzero(3), swab(3)

STANDARDS

The **memset**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

menu_back, **menu_fore**, **menu_grey**, **menu_pad**, **set_menu_back**, **set_menu_fore**, **set_menu_grey**, **set_menu_pad** — get and set menu attributes

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

char
menu_back(MENU *menu);

char
menu_fore(MENU *menu);

char
menu_grey(MENU *menu);

int
menu_pad(MENU *menu);

int
set_menu_back(MENU *menu, char attr);

int
set_menu_fore(MENU *menu, char attr);

int
set_menu_grey(MENU *menu, char attr);

int
set_menu_pad(MENU *menu, int pad);
```

DESCRIPTION

The **menu_back**() function returns the value of the background attribute for the menu passed. This attribute is set by the **set_menu_back**() call. The **menu_fore**() function returns the value of the foreground character attribute for the menu passed. This attribute is set by the **set_menu_fore**() function. The **menu_grey**() function returns the value of the grey or unselectable character attribute for the menu passed. This attribute is set by the **set_menu_grey**() function. The **menu_pad**() function returns the padding character that will be used between the item name and its description. The value of the pad character is set by the **set_menu_pad**() function.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_SYSTEM_ERROR	There was a system error during the call.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.
E_POSTED	The menu is already posted.
E_CONNECTED	An item was already connected to a menu.
E_BAD_STATE	The function was called from within an initialization or termination routine.
E_NO_ROOM	The menu does not fit within the subwindow.
E_NOT_POSTED	The menu is not posted.

E_UNKNOWN_COMMAND	The menu driver does not recognize the request passed to it.
E_NO_MATCH	The character search failed to find a match.
E_NOT_SELECTABLE	The item could not be selected.
E_NOT_CONNECTED	The item is not connected to a menu.
E_REQUEST_DENIED	The menu driver could not process the request.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

pos_menu_cursor — position cursor in menu window

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

int
pos_menu_cursor(MENU *menu);
```

DESCRIPTION

The **pos_menu_cursor**() function positions the cursor in the menu window. This function can be called after other curses calls to restore the cursor to its correct position in the menu.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_SYSTEM_ERROR	There was a system error during the call.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.
E_POSTED	The menu is already posted.
E_CONNECTED	An item was already connected to a menu.
E_BAD_STATE	The function was called from within an initialization or termination routine.
E_NO_ROOM	The menu does not fit within the subwindow.
E_NOT_POSTED	The menu is not posted.
E_UNKNOWN_COMMAND	The menu driver does not recognize the request passed to it.
E_NO_MATCH	The character search failed to find a match.
E_NOT_SELECTABLE	The item could not be selected.
E_NOT_CONNECTED	The item is not connected to a menu.
E_REQUEST_DENIED	The menu driver could not process the request.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

menu_driver — main menu handling function

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

int
menu_driver(MENU *menu, int c);
```

DESCRIPTION

The **menu_driver**() function is the guts of the menu system. It takes the commands passed by c parameter and performs the requested action on the menu given. The following commands may be given to the menu driver:

Command	Action
REQ_LEFT_ITEM	Sets the new current item to be the item to the left of the current item.
REQ_RIGHT_ITEM	Sets the new current item to be the item to the rights of the current item.
REQ_UP_ITEM	Sets the new current item to be the item above the current item.
REQ_DOWN_ITEM	Sets the new current item to be the item below the current item.
REQ_SCR_ULINE	Scroll the menu one line towards the bottom of the menu window. The new current item becomes the item immediately above the current item.
REQ_SCR_DLINE	Scroll the menu one line towards the top of the menu window. The new current item becomes the item immediately below the current item.
REQ_SCR_DPAGE	Scroll the menu one page towards the bottom of the menu window.
REQ_SCR_UPAGE	Scroll the menu one page towards the top of the menu window.
REQ_FIRST_ITEM	Set the current item to be the first item in the menu.
REQ_LAST_ITEM	Set the current item to be the last item in the menu.
REQ_NEXT_ITEM	Set the new current item to be the next item in the item array after the current item.
REQ_PREV_ITEM	Set the new current item to be the item before the current item in the items array.
REQ_TOGGLE_ITEM	If the item is selectable then toggle the item's value.
REQ_CLEAR_PATTERN	Clear all the characters currently in the menu's pattern buffer.
REQ_BACK_PATTERN	Remove the last character from the pattern buffer.
REQ_NEXT_MATCH	Attempt to find the next item that matches the pattern buffer.
REQ_PREV_MATCH	Attempt to find the previous item that matches the pattern buffer.

If **menu_driver**() is passed a command that is greater than MAX_COMMAND then the command passed is assumed to be a user defined command and **menu_driver**() returns E_UNKNOWN_COMMAND. Otherwise if the command is a printable character then the character represented by the command is placed at the end of the pattern buffer and an attempt is made to match the pattern buffer against the items in the menu.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_SYSTEM_ERROR	There was a system error during the call.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.

E_NOT_POSTED	The menu is not posted.
E_UNKNOWN_COMMAND	The menu driver does not recognize the request passed to it.
E_NO_MATCH	The character search failed to find a match.
E_NOT_CONNECTED	The item is not connected to a menu.
E_REQUEST_DENIED	The menu driver could not process the request.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

menu_format, **set_menu_format** — get or set number of rows and columns of items

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

void
menu_format(MENU *menu, int *rows, int *cols);

int
set_menu_format(MENU *menu, int rows, int cols);
```

DESCRIPTION

The **menu_format**() returns the number of rows and columns of items that can be displayed by the menu. The format is set by the **set_menu_format**() function call. Note that the rows and columns defined here are not the size of the window but rather the number of rows and columns of items.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.
E_POSTED	The menu is already posted.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

item_init, **item_term**, **menu_init**, **menu_term**, **set_item_init**, **set_item_term**, **set_menu_init**, **set_menu_term** — get or set handler functions for menu post/unpost or item change

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

void (*hook)()
item_init(MENU *menu);

void (*hook)()
item_term(MENU *menu);

void (*hook)()
menu_init(MENU *menu);

void (*hook)()
menu_term(MENU *menu);

int
set_item_init(MENU *menu, void (*hook)());

int
set_item_term(MENU *menu, void (*hook)());

int
set_menu_init(MENU *menu, void (*hook)());

int
set_menu_term(MENU *menu, void (*hook)());
```

DESCRIPTION

The **item_init**() function returns a pointer to the function that will be called whenever the menu is posted and also just after the current item changes. This is set by the **set_item_init**() call. The **item_term**() function returns a pointer to the function that will be called before the menu is unposted and just before the current item changes, this pointer is set by the **set_item_term**() call. The **menu_init**() functions returns a pointer to the function that will be called just before the menu is posted to the screen. This pointer is set by the **set_menu_init**() function call. The **menu_term**() function returns a pointer to the function that will be called just after the menu has been unposted, this pointer is set by the **set_menu_term**() function.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

current_item, **item_index**, **set_current_item**, **set_top_row** **top_row** — get or set item pointers or top row

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

ITEM *
current_item(MENU *menu);

int
item_index(ITEM *item);

int
set_current_item(MENU *menu, ITEM *item);

int
set_top_row(MENU *menu, int row);

int
top_row(MENU *menu);
```

DESCRIPTION

The **current_item**() returns a pointer to the current menu item. The **set_current_item**() can be used to set this to the item give. The **item_index**() function returns the index number in the array of items for the item pointed to by the *item* parameter. The **set_top_row**() function sets the top row of the menu displayed to be the row given. The current item becomes the leftmost item of the top row. The **top_row**() call returns the row number that is currently at the top of the displayed menu.

RETURN VALUES

current_item() returns NULL if no items are attached to the menu.

E_OK	The function was successful.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.
E_BAD_STATE	The function was called from within an initialization or termination routine.
E_NOT_CONNECTED	The item is not connected to a menu.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

item_description, **item_name** — get item name or description

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

char *
item_description(ITEM *item);

char *
item_name(ITEM *item);
```

DESCRIPTION

The **item_description**() *menu* function returns the description string associated with the passed item. The **item_name**() function returns the name string associated with the passed item.

RETURN VALUES

The function **item_description**() and **item_name**() functions return NULL if the item pointer is not valid.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

free_item, **new_item** — create or delete menu item

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

int
free_item(ITEM *item);

ITEM *
new_item(char *name, char *description);
```

DESCRIPTION

The **free_item**() function destroys the item and frees all allocated storage for that item. The **new_item**() allocates storage for a new item then copies in the item name and description for the new item. A pointer to the newly created item is returned to the caller.

RETURN VALUES

The **new_item**() function returns NULL on failure, the **free_item**() returns one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

item_opts, **item_opts_off**, **item_opts_on** — get or modify options for an item

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

OPTIONS
item_opts(ITEM *item);

int
item_opts_off(ITEM *item, OPTIONS opts);

int
item_opts_on(ITEM *item, OPTIONS opts);
```

DESCRIPTION

The **item_opts**() function returns the options currently set for the given item. The **item_opts_off**() function turns off the options passed in *opts* for the item passed. The **item_opts_on**() function turns on the options passed in *opts* for the item given.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_SYSTEM_ERROR	There was a system error during the call.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

item_userptr, **set_item_userptr** — get or set user pointer for an item

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

char *
item_userptr(ITEM *item);

int
set_item_userptr(ITEM *item, char *userptr);
```

DESCRIPTION

The **item_userptr**() function returns the value of the user defined pointer for the given item, this pointer is defined by the **set_item_userptr**() function.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_SYSTEM_ERROR	There was a system error during the call.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

item_value, **set_item_value**, **item_selected** — get or set value for an item

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

int
item_value(ITEM *item);

int
set_item_value(ITEM *item, int flag);

int
item_selected(MENU *menu, int **array);
```

DESCRIPTION

The **item_value**() function returns value of the item. If the item has been selected then this value will be TRUE. The value can also be set by calling **set_item_value**() to set the value to a defined state. Setting the value to a value other than TRUE or FALSE will have undefined results. The **item_selected**() function returns the number of items that are selected in the menu, that is the number of items whose value is TRUE. The indexes of the selected items will be returned in *array* which will be dynamically allocated to hold the number of indexes. It is the responsibility of the caller to release this storage by calling free(3) when the storage is no longer required. If there are no elements selected in the items array then **item_selected**() will return 0 and *array* will be NULL. If an error occurs **item_selected**() will return one of the below return values which are less than 0.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_NOT_CONNECTED	The item is not connected to a menu.
E_REQUEST_DENIED	The menu driver could not process the request.
E_SYSTEM_ERROR	A system error occurred whilst processing the request.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

The function **item_selected**() is a NetBSD extension and must not be used in portable code.

NAME

item_visible — get visibility status of an item

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

int
item_visible(ITEM *item);
```

DESCRIPTION

The **item_visible**() function returns TRUE if the item passed is currently visible in a menu.

RETURN VALUES

The functions return one of the following error values:

E_BAD_ARGUMENT An incorrect argument was passed to the function.
E_NOT_CONNECTED The item is not connected to a menu.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

item_count, **menu_items**, **set_menu_items** — attach items to menus or check correspondences

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

int
item_count(MENU *menu);

ITEMS **
menu_items(MENU *menu);

int
set_menu_items(MENU *menu, ITEM **items);
```

DESCRIPTION

The **item_count**() *menu* function returns the number of items currently attached to the menu passed. The **menu_items**() function returns a pointer to an array of item pointers that represent the menu items currently attached to the given menu. Apart from using **new_menu**() (see menu_new(3)) menu items may be attached to a menu by calling **set_menu_items**() any items currently attached to the menu will be detached and the NULL terminated array of new items will be attached to the menu.

RETURN VALUES

Any function returning a string pointer will return NULL if an error occurs. Functions returning an integer will return one of the following:

E_OK	The function was successful.
E_POSTED	The menu is already posted.
E_CONNECTED	An item was already connected to a menu.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

menu_mark, **menu_unmark**, **set_menu_mark**, **set_menu_unmark** — get or set strings that show mark status for a menu

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

char *
menu_mark(MENU *menu);

char *
menu_unmark(MENU *menu);

int
set_menu_mark(MENU *menu, char *mark);

int
set_menu_unmark(MENU *menu, char *mark);
```

DESCRIPTION

The **menu_mark**() function returns a pointer to the character string that is used to mark selected items in the menu. The mark string is set by the **set_menu_mark**() function. The **menu_unmark**() function returns a pointer to the character string that is used to indicate a menu items is not selected, this string is set by the **set_menu_unmark**() function. The mark and unmark strings may be of differing lengths, the room allocated to drawing the mark will be the maximum of the lengths of both the mark and unmark strings. The shorter of the two strings will be left justified and space padded.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_SYSTEM_ERROR	There was a system error during the call.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.
E_POSTED	The menu is already posted.
E_CONNECTED	An item was already connected to a menu.
E_BAD_STATE	The function was called from within an initialization or termination routine.
E_NO_ROOM	The menu does not fit within the subwindow.
E_NOT_POSTED	The menu is not posted.
E_UNKNOWN_COMMAND	The menu driver does not recognize the request passed to it.
E_NO_MATCH	The character search failed to find a match.
E_NOT_SELECTABLE	The item could not be selected.
E_NOT_CONNECTED	The item is not connected to a menu.
E_REQUEST_DENIED	The menu driver could not process the request.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

free_menu, **new_menu** — create or delete a menu

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

int
free_menu(MENU *menu);

MENU *
new_menu(ITEMS **items);
```

DESCRIPTION

The **free_menu**() *menu* function destroys the given menu and frees all allocated storage associated with the menu. All items associated with the menu are detached from the menu before it is destroyed. The **new_menu**() function allocates storage for a new menu and initializes all the values to the defined defaults. If the items pointer passed is not a NULL then the given NULL terminated array of items is attached to the new menu.

RETURN VALUES

The **new_menu**() function returns NULL on error, while the **free_menu**() function returns one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.
E_POSTED	The menu is already posted.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

menu_opts, **menu_opts_off**, **menu_opts_on**, **set_menu_opts** — get or modify options for a menu

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

OPTIONS
menu_opts(MENU *menu);

int
menu_opts_off(MENU *menu, OPTIONS opts);

int
menu_opts_on(MENU *menu, OPTIONS opts);

int
set_menu_opts(MENU *menu, OPTIONS opts);
```

DESCRIPTION

The **menu_opts**() function returns the current options set for the menu given. The **menu_opts_off**() function turns off the menu options given by the opts parameter for the menu. The **menu_opts_on**() function turns on the menu options given by the opts parameter for the menu passed. The **set_menu_opts**() sets the menu options to the value given in opts.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.
E_POSTED	The menu is already posted.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

menu_pattern, **set_menu_pattern** — get or set menu pattern

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

char *
menu_pattern(MENU *menu);
```

DESCRIPTION

The **menu_pattern**() function returns a pointer to the string that is currently in the menu pattern buffer. The menu pattern buffer can be set by calling **set_menu_pattern**() which will set the pattern buffer to the string passed and then attempt to match that string against the names of the items in the attached items.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_SYSTEM_ERROR	There was a system error during the call.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.
E_NO_MATCH	The character search failed to find a match.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

post_menu **unpost_menu** — post (draw) or unpost a menu

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

int
post_menu(MENU *menu);

int
unpost_menu(MENU *menu);
```

DESCRIPTION

The **post_menu**() function causes the menu to be drawn on the screen. Any functions defined by either **set_menu_init**() or **set_item_init**() (see menu_hook(3)) are called before the menu is placed on the screen. The **unpost_menu**() does the opposite, it removes a menu from the screen. Any functions defined by both **set_menu_term**() and **set_item_term**() (see menu_hook(3)) are called prior to the menu's removal.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_SYSTEM_ERROR	There was a system error during the call.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.
E_POSTED	The menu is already posted.
E_BAD_STATE	The function was called from within an initialization or termination routine.
E_NO_ROOM	The menu does not fit within the subwindow.
E_NOT_CONNECTED	The item is not connected to a menu.

SEE ALSO

curses(3), menu_hook(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

menu_userptr, **set_menu_userptr** — get or set user pointer for a menu

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

char *
menu_userptr(MENU *menu);

int
set_menu_userptr(MENU *menu, char *ptr);
```

DESCRIPTION

The **menu_userptr**() function returns the pointer to the user defined pointer for the given menu. This pointer is set by the **set_menu_userptr**() function.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_SYSTEM_ERROR	There was a system error during the call.

SEE ALSO

curses(3), menus(3)

NOTES

The header <menu.h> automatically includes both <curses.h> and <eti.h>.

NAME

menu_sub, **menu_win**, **scale_menu**, **set_menu_sub**, **set_menu_win** — sub-menu handling

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>

WINDOW *
menu_sub(MENU *menu);

WINDOW *
menu_win(MENU *menu);

int
scale_menu(MENU *menu, int *rows, int *cols);

int
set_menu_sub(MENU *menu, WINDOW *sub);

int
set_menu_win(MENU *menu, WINDOW *win);
```

DESCRIPTION

The **menu_sub**() function returns a pointer to the window that will be used to post a menu into, this pointer is set by the **set_menu_sub**() function. The **menu_win**() function returns a pointer to the window in which the menu subwindow will be created when the menu is posted, this pointer is set by the **set_menu_win**() function. The **scale_menu**() function calculates the minimum size a window needs to be to hold the items for a given menu, the parameters rows and cols are set to the required number of rows and columns respectively.

RETURN VALUES

The functions return one of the following error values:

E_OK	The function was successful.
E_SYSTEM_ERROR	There was a system error during the call.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.
E_POSTED	The menu is already posted.
E_CONNECTED	An item was already connected to a menu.
E_BAD_STATE	The function was called from within an initialization or termination routine.
E_NO_ROOM	The menu does not fit within the subwindow.
E_NOT_POSTED	The menu is not posted
E_UNKNOWN_COMMAND	The menu driver does not recognize the request passed to it.
E_NO_MATCH	The character search failed to find a match.
E_NOT_SELECTABLE	The item could not be selected.
E_NOT_CONNECTED	The item is not connected to a menu.
E_REQUEST_DENIED	The menu driver could not process the request.

SEE ALSO

curses(3), menus(3)

NOTES

The header `<menu.h>` automatically includes both `<curses.h>` and `<eti.h>`.

NAME

menus — menu library

LIBRARY

Curses Menu Library (libmenu, −lmenu)

SYNOPSIS

```
#include <menu.h>
```

DESCRIPTION

The **menus** library provides a terminal independent menu system using the curses(3) library. Before using the **menus** functions the terminal must be set up by curses(3) using the **initscr**() function or similar. Programs using **menus** functions must be linked with the curses(3) library.

The **menus** library provides facilities for defining menu items, placing a menu on the terminal screen, assign pre- and post-change operations and setting the attributes of both the menu and its items.

Defining default attributes for menus and items

The **menus** library allows any settable attribute or option of both the menu and item objects to be defined such that any new menu or item automatically inherits the value as default. Setting the default value will not affect any item or menu that has already been created but will be applied to subsequent objects. To set the default attribute or option the set routine is passed a NULL pointer in the item or menu parameter when calling the set routine. The current default value can be retrieved by calling the get routine with a NULL pointer for the item or menu parameter.

function name	manual page name
current_item	menu_item_current(3)
free_item	menu_item_new(3)
free_menu	menu_new(3)
item_count	menu_items(3)
item_description	menu_item_name(3)
item_index	menu_item_current(3)
item_init	menu_hook(3)
item_name	menu_item_name(3)
item_opts	menu_item_opts(3)
item_opts_off	menu_item_opts(3)
item_opts_on	menu_item_opts(3)
item_selected	menu_item_value(3)
item_term	menu_hook(3)
item_userptr	menu_item_userptr(3)
item_value	menu_item_value(3)
item_visible	menu_item_visible(3)
menu_back	menu_attributes(3)
menu_driver	menu_driver(3)
menu_fore	menu_attributes(3)
menu_format	menu_format(3)
menu_grey	menu_attributes(3)
menu_init	menu_hook(3)
menu_items	menu_items(3)
menu_mark	menu_mark(3)

menu_opts	menu_opts(3)
menu_opts_off	menu_opts(3)
menu_opts_on	menu_opts(3)
menu_pad	menu_attributes(3)
menu_pattern	menu_pattern(3)
menu_sub	menu_win(3)
menu_term	menu_hook(3)
menu_unmark	menu_mark(3)
menu_userptr	menu_userptr(3)
men_win	menu_win(3)
new_item	menu_item_new(3)
new_menu	menu_new(3)
pos_menu_cursor	menu_cursor(3)
post_menu	menu_post(3)
scale_menu	menu_win(3)
set_current_item	menu_item_current(3)
set_item_init	menu_hook(3)
set_item_opts	menu_item_opts(3)
set_item_term	menu_hook(3)
set_item_userptr	menu_item_userptr(3)
set_item_value	menu_item_value(3)
set_menu_back	menu_attributes(3)
set_menu_fore	menu_attributes(3)
set_menu_format	menu_format(3)
set_menu_grey	menu_attributes(3)
set_menu_init	menu_hook(3)
set_menu_items	menu_items(3)
set_menu_mark	menu_mark(3)
set_menu_opts	menu_opts(3)
set_menu_pad	menu_attributes(3)
set_menu_pattern	menu_pattern(3)
set_menu_sub	menu_win(3)
set_menu_term	menu_hook(3)
set_menu_unmark	menu_mark(3)
set_menu_userptr	menu_userptr(3)
set_menu_win	menu_win(3)
set_top_row	menu_item_current(3)
top_row	menu_item_current(3)
unpost_menu	menu_post(3)

RETURN VALUES

Any function returning a string pointer will return NULL if an error occurs. Functions returning an integer will return one of the following:

E_OK	The function was successful.
E_SYSTEM_ERROR	There was a system error during the call.
E_BAD_ARGUMENT	One or more of the arguments passed to the function was incorrect.
E_POSTED	The menu is already posted.
E_CONNECTED	An item was already connected to a menu.
E_BAD_STATE	The function was called from within an initialization or termination routine.

E_NO_ROOM	The menu does not fit within the subwindow.
E_NOT_POSTED	The menu is not posted.
E_UNKNOWN_COMMAND	The menu driver does not recognize the request passed to it.
E_NO_MATCH	The character search failed to find a match.
E_NOT_SELECTABLE	The item could not be selected.
E_NOT_CONNECTED	The item is not connected to a menu.
E_REQUEST_DENIED	The menu driver could not process the request.

SEE ALSO

curses(3), menu_attributes(3), menu_cursor(3), menu_driver(3), menu_format(3), menu_hook(3), menu_item_current(3), menu_item_name(3), menu_item_new(3), menu_item_opts(3), menu_item_userptr(3), menu_item_value(3), menu_item_visible(3), menu_items(3), menu_mark(3), menu_new(3), menu_opts(3), menu_pattern(3), menu_post(3), menu_userptr(3), menu_win(3)

NOTES

This implementation of the menus library does depart in behaviour subtly from the original AT & T implementation. Some of the more notable departures are:

unmark — The original implementation did not have a marker for an unmarked field the mark was only displayed next to a field when it had been marked using the REQ_TOGGLE_ITEM. In this implementation a separate marker can be used to indicate an unmarked item. This can be set using set_menu_unmark function. There is no requirement for the mark and unmark strings to be the same length. Room will be left for the longest of the two. The unmark string is optional, if it is not set then menus defaults to the old behaviour.

item marking — In the original implementation the current item was considered selected and hence had the mark string displayed next to it. This implementation does not do this because the Author considers the effect too confusing. Especially in the case of a multiple selection menu because there was no way to tell if the current item is selected or not without shifting off of it. Since the current item is displayed using the foreground attribute it was deemed unnecessary to also display the mark string against the current item.

The option O_RADIO and the function **item_selected**() are NetBSD extensions and must not be used in portable code.

NAME

mktemp, **mkstemp**, **mkdtemp** — make unique temporary file or directory name

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

char *
mktemp(char *template);

int
mkstemp(char *template);

char *
mkdtemp(char *template);
```

DESCRIPTION

The **mktemp**() function takes the given file name template and overwrites a portion of it to create a file name. This file name is unique and suitable for use by the application. The template may be any file name with some number of 'X's appended to it, for example /tmp/temp.XXXXXX. The trailing 'X's are replaced with the current process number and/or a unique letter combination. The number of unique file names **mktemp**() can return depends on the number of 'X's provided. Although the NetBSD implementation of the functions will accept any number of trailing 'X's, for portability reasons one should use only six. Using six 'X's will result in **mktemp**() testing roughly 26 ** 6 (308915776) combinations.

The **mkstemp**() function makes the same replacement to the template and creates the template file, mode 0600, returning a file descriptor opened for reading and writing. This avoids the race between testing for a file's existence and opening it for use.

The **mkdtemp**() function is similar to **mkstemp**(), but it creates a mode 0700 directory instead and returns the path.

Please note that the permissions of the file or directory being created are subject to the restrictions imposed by the umask(2) system call. It may thus happen that the created file is unreadable and/or unwritable.

RETURN VALUES

The **mktemp**() and **mkdtemp**() functions return a pointer to the template on success and NULL on failure. The **mkstemp**() function returns −1 if no suitable file could be created. If either call fails an error code is placed in the global variable *errno*.

EXAMPLES

Quite often a programmer will want to replace a use of **mktemp**() with **mkstemp**(), usually to avoid the problems described above. Doing this correctly requires a good understanding of the code in question.

For instance, code of this form:

```
char sfn[15] = "";
FILE *sfp;

strlcpy(sfn, "/tmp/ed.XXXXXX", sizeof sfn);
if (mktemp(sfn) == NULL || (sfp = fopen(sfn, "w+")) == NULL) {
        fprintf(stderr, "%s: %s\n", sfn, strerror(errno));
        return (NULL);
}
```

```
return (sfp);
```

should be rewritten like this:

```
char sfn[15] = "";
FILE *sfp;
int fd = -1;

strlcpy(sfn, "/tmp/ed.XXXXXX", sizeof sfn);
if ((fd = mkstemp(sfn)) == -1 ||
    (sfp = fdopen(fd, "w+")) == NULL) {
        if (fd != -1) {
                unlink(sfn);
                close(fd);
        }
        fprintf(stderr, "%s: %s\n", sfn, strerror(errno));
        return (NULL);
}
return (sfp);
```

Often one will find code which uses **mktemp**() very early on, perhaps to globally initialize the template nicely, but the code which calls open(2) or fopen(3) on that filename will occur much later. (In almost all cases, the use of fopen(3) will mean that the flags O_CREAT | O_EXCL are not given to open(2), and thus a symbolic link race becomes possible, hence making necessary the use of fdopen(3) as seen above). Furthermore, one must be careful about code which opens, closes, and then re-opens the file in question. Finally, one must ensure that upon error the temporary file is removed correctly.

There are also cases where modifying the code to use **mktemp**(), in concert with open(2) using the flags O_CREAT | O_EXCL, is better, as long as the code retries a new template if open(2) fails with an *errno* of EEXIST.

ERRORS

The **mktemp**(), **mkstemp**() and **mkdtemp**() functions may set *errno* to one of the following values:

[ENOTDIR] The pathname portion of the template is not an existing directory.

The **mktemp**(), **mkstemp**() and **mkdtemp**() functions may also set *errno* to any value specified by the stat(2) function.

The **mkstemp**() function may also set *errno* to any value specified by the open(2) function.

The **mkdtemp**() function may also set *errno* to any value specified by the mkdir(2) function.

SEE ALSO

chmod(2), getpid(2), open(2), stat(2), umask(2)

HISTORY

A **mktemp**() function appeared in Version 7 AT&T UNIX.

The **mkstemp**() function appeared in 4.4BSD.

The **mkdtemp**() function appeared in NetBSD 1.4.

BUGS

For **mktemp**() there is an obvious race between file name selection and file creation and deletion: the program is typically written to call tmpnam(3), tempnam(3), or **mktemp**(). Subsequently, the program calls open(2) or fopen(3) and erroneously opens a file (or symbolic link, fifo or other device) that the attacker

has created in the expected file location. Hence **mkstemp**() is recommended, since it atomically creates the file. An attacker can guess the filenames produced by **mktemp**(). Whenever it is possible, **mkstemp**() or **mkdtemp**() should be used instead.

For this reason, ld(1) will output a warning message whenever it links code that uses **mktemp**().

The **mkdtemp**() function is nonstandard and should not be used if portability is required.

SECURITY CONSIDERATIONS

The use of **mktemp**() should generally be avoided, as a hostile process can exploit a race condition in the time between the generation of a temporary filename by **mktemp**() and the invoker's use of the temporary name. A link-time warning will be issued advising the use of **mkstemp**() or **mkdtemp**() instead.

NAME

modf — extract signed integral and fractional values from floating-point number

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
modf(double value, double *iptr);

float
modff(float value, float *iptr);
```

DESCRIPTION

The **modf**() function breaks the argument *value* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a *double* in the object pointed to by *iptr*.

RETURN VALUES

The **modf**() function returns the signed fractional part of *value*.

SEE ALSO

frexp(3), ldexp(3), math(3)

STANDARDS

The **modf**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

moncontrol, **monstartup** — control execution profile

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

moncontrol(*int mode*);

monstartup(*u_long *lowpc*, *u_long *highpc*);

DESCRIPTION

An executable program compiled using the **-pg** option to cc(1) automatically includes calls to collect statistics for the gprof(1) call-graph execution profiler. In typical operation, profiling begins at program startup and ends when the program calls exit. When the program exits, the profiling data are written to the file *gmon.out*, then gprof(1) can be used to examine the results.

moncontrol() selectively controls profiling within a program. When the program starts, profiling begins. To stop the collection of histogram ticks and call counts use **moncontrol**(*0*); to resume the collection of histogram ticks and call counts use **moncontrol**(*1*). This feature allows the cost of particular operations to be measured. Note that an output file will be produced on program exit regardless of the state of **moncontrol**().

Programs that are not loaded with **-pg** may selectively collect profiling statistics by calling **monstartup**() with the range of addresses to be profiled. *lowpc* and *highpc* specify the address range that is to be sampled; the lowest address sampled is that of *lowpc* and the highest is just below *highpc*. Only functions in that range that have been compiled with the **-pg** option to cc(1) will appear in the call graph part of the output; however, all functions in that address range will have their execution time measured. Profiling begins on return from **monstartup**().

ENVIRONMENT

PROFDIR Directory to place the output file(s) in. When this is set, instead of writing the profiling output to gmon.out, a filename is generated from the process id and name of the program (e.g., 123.a.out). If you are profiling a program that forks, or otherwise creates multiple copies, setting this is the only reasonable way to get all profiling data.

FILES

gmon.out execution data file

SEE ALSO

cc(1), gprof(1), profil(2)

NAME

mpool, **mpool_open**, **mpool_filter**, **mpool_new**, **mpool_get**, **mpool_put**, **mpool_sync**, **mpool_close** — shared memory buffer pool

SYNOPSIS

```
#include <db.h>
#include <mpool.h>

MPOOL *
mpool_open(DBT *key, int fd, pgno_t pagesize, pgno_t maxcache);

void
mpool_filter(MPOOL *mp, void (*pgin)(void *, pgno_t, void *),
    void (*pgout)(void *, pgno_t, void *), void *pgcookie);

void *
mpool_new(MPOOL *mp, pgno_t *pgnoaddr);

void *
mpool_get(MPOOL *mp, pgno_t pgno, u_int flags);

int
mpool_put(MPOOL *mp, void *pgaddr, u_int flags);

int
mpool_sync(MPOOL *mp);

int
mpool_close(MPOOL *mp);
```

DESCRIPTION

mpool is the library interface intended to provide page oriented buffer management of files. The buffers may be shared between processes.

The function **mpool_open** initializes a memory pool. The *key* argument is the byte string used to negotiate between multiple processes wishing to share buffers. If the file buffers are mapped in shared memory, all processes using the same key will share the buffers. If *key* is NULL, the buffers are mapped into private memory. The *fd* argument is a file descriptor for the underlying file, which must be seekable. If *key* is non-NULL and matches a file already being mapped, the *fd* argument is ignored.

The *pagesize* argument is the size, in bytes, of the pages into which the file is broken up. The *maxcache* argument is the maximum number of pages from the underlying file to cache at any one time. This value is not relative to the number of processes which share a file's buffers, but will be the largest value specified by any of the processes sharing the file.

The **mpool_filter** function is intended to make transparent input and output processing of the pages possible. If the *pgin* function is specified, it is called each time a buffer is read into the memory pool from the backing file. If the *pgout* function is specified, it is called each time a buffer is written into the backing file. Both functions are are called with the *pgcookie* pointer, the page number and a pointer to the page to being read or written.

The function **mpool_new** takes an MPOOL pointer and an address as arguments. If a new page can be allocated, a pointer to the page is returned and the page number is stored into the *pgnoaddr* address. Otherwise, NULL is returned and errno is set.

The function **mpool_get** takes a MPOOL pointer and a page number as arguments. If the page exists, a pointer to the page is returned. Otherwise, NULL is returned and errno is set. The flags parameter is not currently used.

The function **mpool_put** unpins the page referenced by *pgaddr*. *pgaddr* must be an address previously returned by **mpool_get** or **mpool_new**. The flag value is specified by or'ing any of the following values:

MPOOL_DIRTY The page has been modified and needs to be written to the backing file.

mpool_put returns 0 on success and −1 if an error occurs.

The function **mpool_sync** writes all modified pages associated with the MPOOL pointer to the backing file. **mpool_sync** returns 0 on success and −1 if an error occurs.

The **mpool_close** function frees up any allocated memory associated with the memory pool cookie. Modified pages are *not* written to the backing file. **mpool_close** returns 0 on success and −1 if an error occurs.

ERRORS

The **mpool_open** function may fail and set *errno* for any of the errors specified for the library routine malloc(3).

The **mpool_get** function may fail and set *errno* for the following:

EINVAL The requested record doesn't exist.

The **mpool_new** and **mpool_get** functions may fail and set *errno* for any of the errors specified for the library routines read(2), write(2), and malloc(3).

The **mpool_sync** function may fail and set *errno* for any of the errors specified for the library routine write(2).

The **mpool_close** function may fail and set *errno* for any of the errors specified for the library routine free(3).

SEE ALSO

btree(3), dbopen(3), hash(3), recno(3)

NAME

nan, **nanf**, **nanl** — return quiet NaN

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
nan(const char *tagp);

float
nanf(const char *tagp);

long double
nanl(const char *tagp);
```

DESCRIPTION

The call **nan**(*"n-char-sequence"*) is equivalent to the call **strod**(*"NAN(n-char-sequence)"*, *NULL*). The call **nan**(*""*) is equivalent to the call **strod**(*"NAN()"*, *NULL*).

The **nanf**() and **nanl**() functions are equivalent to **nan**() but substituting **strtof**() and **strtod**(), respectively.

RETURN VALUES

IEEE 754

The **nan**(), **nanf**(), and **nanl**() functions return a quiet NaN as specified by *tagp*.

VAX

The **nan**(), **nanf**(), and **nanl**() functions return zero.

SEE ALSO

math(3), strod(3)

STANDARDS

The **nan**(), **nanf**(), and **nanl**() functions conform to ISO/IEC 9899:1999 (“ISO C99”).

NAME

ngettext, dngettext, dcngettext − translate message and choose plural form

SYNOPSIS

#include <libintl.h>

char * ngettext (const char * *msgid***, const char *** *msgid_plural***,**
unsigned long int *n***);**
char * dngettext (const char * *domainname***,**
const char * *msgid***, const char *** *msgid_plural***,**
unsigned long int *n***);**
char * dcngettext (const char * *domainname***,**
const char * *msgid***, const char *** *msgid_plural***,**
unsigned long int *n***, int** *category***);**

DESCRIPTION

The **ngettext**, **dngettext** and **dcngettext** functions attempt to translate a text string into the user's native language, by looking up the appropriate plural form of the translation in a message catalog.

Plural forms are grammatical variants depending on the a number. Some languages have two forms, called singular and plural. Other languages have three forms, called singular, dual and plural. There are also languages with four forms.

The **ngettext**, **dngettext** and **dcngettext** functions work like the **gettext**, **dgettext** and **dcgettext** functions, respectively. Additionally, they choose the appropriate plural form, which depends on the number *n* and the language of the message catalog where the translation was found.

In the "C" locale, or if none of the used catalogs contain a translation for *msgid*, the **ngettext**, **dngettext** and **dcngettext** functions return *msgid* if *n* == 1, or *msgid_plural* if *n* != 1.

RETURN VALUE

If a translation was found in one of the specified catalogs, the appropriate plural form is converted to the locale's codeset and returned. The resulting string is statically allocated and must not be modified or freed. Otherwise *msgid* or *msgid_plural* is returned, as described above.

ERRORS

errno is not modified.

BUGS

The return type ought to be **const char ***, but is **char *** to avoid warnings in C code predating ANSI C.

SEE ALSO

gettext(3), **dgettext**(3), **dcgettext**(3)

NAME

nice — set program scheduling priority

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
nice(int incr);
```

DESCRIPTION

This interface is obsoleted by `setpriority`**(2).**

The **nice**() function obtains the scheduling priority of the process from the system and sets it to the priority value specified in *incr*. The priority is a value in the range -20 to 20. The default priority is 0; lower priorities cause more favorable scheduling. Only the super-user may lower priorities.

Children inherit the priority of their parent processes via `fork`(2).

RETURN VALUES

Upon successful completion, **nice**() returns the new nice value minus `NZERO`. Otherwise, −1 is returned, the process' nice value is not changed, and *errno* is set to indicate the error.

ERRORS

The **nice**() function will fail if:

`[EPERM]` The *incr* argument is negative and the caller is not the super-user.

SEE ALSO

`nice`(1), `fork`(2), `setpriority`(2), `renice`(8)

STANDARDS

The **nice**() function conforms to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

HISTORY

A **nice**() syscall appeared in Version 6 AT&T UNIX.

NAME

nl_langinfo — get locale information

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <langinfo.h>

char *
nl_langinfo(nl_item item);
```

DESCRIPTION

The **nl_langinfo**() function returns a pointer to a string containing information set by the program's locale.

The names and values of *item* are defined in ⟨langinfo.h⟩. The entries under Category indicate in which setlocale(3) category each item is defined.

Constant	*Category*	*Meaning*
CODESET	LC_CTYPE	Codeset name
D_T_FMT	LC_TIME	String for formatting date and time
D_FMT	LC_TIME	Date format string
T_FMT	LC_TIME	Time format string
T_FMT_AMPM	LC_TIME	A.M. or P.M. time format string
AM_STR	LC_TIME	Ante-meridiem affix
PM_STR	LC_TIME	Post-meridiem affix
DAY_1	LC_TIME	Name of the first day of the week (e.g.: Sunday)
DAY_2	LC_TIME	Name of the second day of the week (e.g.: Monday)
DAY_3	LC_TIME	Name of the third day of the week (e.g.: Tuesday)
DAY_4	LC_TIME	Name of the fourth day of the week (e.g.: Wednesday)
DAY_5	LC_TIME	Name of the fifth day of the week (e.g.: Thursday)
DAY_6	LC_TIME	Name of the sixth day of the week (e.g.: Friday)
DAY_7	LC_TIME	Name of the seventh day of the week (e.g.: Saturday)
ABDAY_1	LC_TIME	Abbreviated name of the first day of the week
ABDAY_2	LC_TIME	Abbreviated name of the second day of the week
ABDAY_3	LC_TIME	Abbreviated name of the third day of the week
ABDAY_4	LC_TIME	Abbreviated name of the fourth day of the week
ABDAY_5	LC_TIME	Abbreviated name of the fifth day of the week
ABDAY_6	LC_TIME	Abbreviated name of the sixth day of the week
ABDAY_7	LC_TIME	Abbreviated name of the seventh day of the week
MON_1	LC_TIME	Name of the first month of the year
MON_2	LC_TIME	Name of the second month
MON_3	LC_TIME	Name of the third month
MON_4	LC_TIME	Name of the fourth month
MON_5	LC_TIME	Name of the fifth month
MON_6	LC_TIME	Name of the sixth month
MON_7	LC_TIME	Name of the seventh month
MON_8	LC_TIME	Name of the eighth month
MON_9	LC_TIME	Name of the ninth month
MON_10	LC_TIME	Name of the tenth month
MON_11	LC_TIME	Name of the eleventh month
MON_12	LC_TIME	Name of the twelfth month

ABMON_1	LC_TIME	Abbreviated name of the first month
ABMON_2	LC_TIME	Abbreviated name of the second month
ABMON_3	LC_TIME	Abbreviated name of the third month
ABMON_4	LC_TIME	Abbreviated name of the fourth month
ABMON_5	LC_TIME	Abbreviated name of the fifth month
ABMON_6	LC_TIME	Abbreviated name of the sixth month
ABMON_7	LC_TIME	Abbreviated name of the seventh month
ABMON_8	LC_TIME	Abbreviated name of the eighth month
ABMON_9	LC_TIME	Abbreviated name of the ninth month
ABMON_10	LC_TIME	Abbreviated name of the tenth month
ABMON_11	LC_TIME	Abbreviated name of the eleventh month
ABMON_12	LC_TIME	Abbreviated name of the twelfth month
ERA	LC_TIME	Era description segments
ERA_D_FMT	LC_TIME	Era date format string
ERA_D_T_FMT	LC_TIME	Era date and time format string
ERA_T_FMT	LC_TIME	Era time format string
ALT_DIGITS	LC_TIME	Alternative symbols for digits
RADIXCHAR	LC_NUMERIC	Radix character
THOUSEP	LC_NUMERIC	Separator for thousands
YESEXPR	LC_MESSAGES	Affirmative response expression
NOEXPR	LC_MESSAGES	Negative response expression

RETURN VALUES

nl_langinfo() returns a pointer to an empty string if *item* is invalid.

EXAMPLES

The following example uses **nl_langinfo**() to obtain the date and time format for the current locale:

```
#include <time.h>
#include <langinfo.h>
#include <locale.h>

int main(void)
{
        char datestring[100];
        struct tm *tm;
        time_t t;
        char *ptr;

        t = time(NULL);
        tm = localtime(&t);
        (void)setlocale(LC_ALL, "");
        ptr = nl_langinfo(D_T_FMT);
        strftime(datestring, sizeof(datestring), ptr, tm);
        printf("%s\n",datestring);
        return (0);
}
```

SEE ALSO

setlocale(3), nls(7)

STANDARDS

The **nl_langinfo**() function conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

HISTORY

The **nl_langinfo**() function appeared in NetBSD 1.0.

NAME

nlist — retrieve symbol table name list from an executable file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <nlist.h>

int
nlist(const char *filename, struct nlist *nl);
```

DESCRIPTION

The **nlist**() function retrieves name list entries from the symbol table of an executable file. (See a.out(5).) The argument *nl* is set to reference the beginning of the list. The list is preened of binary and invalid data; if an entry in the name list is valid, the *n_type* and *n_value* for the entry are copied into the list referenced by *nl*. No other data is copied. The last entry in the list is always NULL.

RETURN VALUES

The number of invalid entries is returned if successful; otherwise, if the file *filename* does not exist or is not executable, the returned value is −1.

SEE ALSO

a.out(5)

HISTORY

A **nlist**() function appeared in Version 6 AT&T UNIX.

NAME

nsdispatch — name-service switch dispatcher routine

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <nsswitch.h>

int
nsdispatch(void *nsdrv, const ns_dtab dtab[], const char *database,
    const char *name, const ns_src defaults[], ...);
```

DESCRIPTION

The **nsdispatch**() function invokes the callback functions specified in *dtab* in the order given in /etc/nsswitch.conf for the database *database* until the action criteria for a source of that database is fulfilled.

nsdrv is passed to each callback function to use as necessary (to pass back to the caller of **nsdispatch**()).

dtab is an array of *ns_dtab* structures, which have the following format:

```
typedef struct {
        const char *src;
        nss_method cb;
        void *cb_data;
} ns_dtab;
```

The *dtab* array should consist of one entry for each source type that has a static implementation, with *src* as the name of the source, *cb* as a callback function which handles that source, and *cb_data* as a pointer to arbitrary data to be passed to the callback function. The last entry in *dtab* should contain NULL values for *src*, *cb*, and *cb_data*.

The callback function signature is described by the typedef:

typedef int (***nss_method**)(*void *cbrv*, *void *cbdata*, *va_list ap*);

cbrv The *nsdrv* that **nsdispatch**() was invoked with.

cbdata The *cb_data* member of the array entry for the source that this callback function implements in the *dtab* argument of **nsdispatch**().

ap The ... arguments to **nsdispatch**(), converted to a *va_list*.

database and *name* are used to select methods from optional per-source dynamically-loaded modules. *name* is usually the name of the function calling **nsdispatch**(). Note that the callback functions provided by *dtab* take priority over those implemented in dynamically-loaded modules in the event of a conflict.

defaults contains a list of default sources to try in the case of a missing or corrupt nsswitch.conf(5), or if there isn't a relevant entry for *database*. It is an array of *ns_src* structures, which have the following format:

```
typedef struct {
        const char *src;
        uint32_t flags;
} ns_src;
```

The *defaults* array should consist of one entry for each source to consult by default indicated by *src*, and *flags* set to the desired behavior (usually NS_SUCCESS; refer to **Callback function return values** for more information). The last entry in *defaults* should have *src* set to NULL and *flags* set to 0.

Some invokers of **nsdispatch**() (such as setgrent(3)) need to force all callback functions to be invoked, irrespective of the action criteria listed in nsswitch.conf(5). This can be achieved by adding NS_FORCEALL to *defaults[0].flags* before invoking **nsdispatch**(). The return value of **nsdispatch**() will be the result of the final callback function invoked.

For convenience, a global variable defined as:

```
extern const ns_src __nsdefaultsrc[];
```

exists which contains a single default entry for 'files' for use by callers which don't require complicated default rules.

`...` are optional extra arguments, which are passed to the appropriate callback function as a stdarg(3) variable argument list of the type *va_list*.

nsdispatch returns the value of the callback function that caused the dispatcher to finish, or NS_NOTFOUND otherwise.

Dynamically-loaded module interface

The **nsdispatch**() function loads callback functions from the run-time link-editor's search path using the following naming convention:

nss_<source>.so.<version>

⟨source⟩ The source that the module implements.

⟨version⟩ The **nsdispatch** module interface version, which is defined by the integer NSS_MODULE_INTERFACE_VERSION, which has the value 0.

When a module is loaded, **nsdispatch**() looks for and calls the following function in the module:

ns_mtab ∗**nss_module_register**(*const char* ∗*source*, *u_int* ∗*nelems*, *nss_module_unregister_fn* ∗*unreg*);

source The name of the source that the module implements, as used by **nsdispatch**() to construct the module's name.

nelems A pointer to an unsigned integer that **nss_module_register**() should set to the number of elements in the *ns_mtab* array returned by **nss_module_register**(), or 0 if there was a failure.

unreg A pointer to a function pointer that **nss_module_register**() can optionally set to an unregister function to be invoked when the module is unloaded, or NULL if there isn't one.

The unregister function signature is described by the typedef:

typedef void (∗**nss_module_unregister_fn**)(*ns_mtab* ∗*mtab*, *u_int nelems*);

mtab The array of *ns_mtab* structures returned by **nss_module_register**().

nelems The ∗*nelems* value set by **nss_module_register**().

nss_module_register() returns an array of *ns_mtab* structures (with ∗*nelems* entries), or NULL if there was a failure. The *ns_mtab* structures have the following format:

```
typedef struct {
        const char *database;
        const char *name;
        nss_method method;
        void *mdata;
} ns_mtab;
```

The *mtab* array should consist of one entry for each callback function (method) that is implemented, with *database* as the name of the database, *name* as the name of the callback function, *method* as the *nss_method* callback function that implements the method, and *mdata* as a pointer to arbitrary data to be passed to the callback function as its *cbdata* argument.

Valid source types

While there is support for arbitrary sources, the following #defines for commonly implemented sources are provided:

#define	Value
NSSRC_FILES	"files"
NSSRC_DNS	"dns"
NSSRC_NIS	"nis"
NSSRC_COMPAT	"compat"

Refer to `nsswitch.conf`(5) for a complete description of what each source type is.

Valid database types

While there is support for arbitrary databases, the following #defines for currently implemented system databases are provided:

#define	Value
NSDB_HOSTS	"hosts"
NSDB_GROUP	"group"
NSDB_GROUP_COMPAT	"group_compat"
NSDB_NETGROUP	"netgroup"
NSDB_NETWORKS	"networks"
NSDB_PASSWD	"passwd"
NSDB_PASSWD_COMPAT	"passwd_compat"
NSDB_SHELLS	"shells"

Refer to `nsswitch.conf`(5) for a complete description of what each database is.

Callback function return values

The callback functions should return one of the following values depending upon status of the lookup:

Return value	Status code
NS_SUCCESS	The requested entry was found.
NS_NOTFOUND	The entry is not present at this source.
NS_TRYAGAIN	The source is busy, and may respond to retries.
NS_UNAVAIL	The source is not responding, or entry is corrupt.

CALLBACK FUNCTION API FOR STANDARD DATABASES

The organization of the *ap* argument for an **nss_method**() callback function for a standard method in a standard database is:

1. Pointer to return value of the standard function.
2. First argument of the standard function.
3. (etc.)

For example, given the standard function getgrnam(3):

struct group * **getgrnam**(*const char* **name*)

the *ap* organization used by the callback functions is:

1. *struct group* **
2. *const char* *

NOTE: Not all standard databases are using this calling convention yet; those that aren't are noted below. These will be changed in the future.

The callback function names and *va_list* organization for various standard database callback functions are:

Methods for hosts database

NOTE: The method APIs for this database will be changing in the near future.

getaddrinfo

char **name*, *const struct addrinfo* **pai*

Returns *struct addrinfo* * via *void* **cbrv*.

gethostbyaddr

unsigned char **addr*, *int addrlen*, *int af*

Returns *struct hostent* * via *void* **cbrv*.

gethostbyname

char **name*, *int namelen*, *int af*

Returns *struct hostent* * via *void* **cbrv*.

Methods for group and group_compat databases

endgrent

Empty *ap*.

All methods for all sources are invoked for this method name.

getgrent

struct group ***retval*

**retval* should be set to a pointer to an internal static *struct group* on success, NULL otherwise.

getgrent(3) returns **retval* if **nsdispatch**() returns NS_SUCCESS, NULL otherwise.

getgrent_r

int **retval*, *struct group* **grp*, *char* **buffer*, *size_t buflen*, *struct group* ***result*

**retval* should be set to an appropriate errno(2) on failure.

getgrent_r(3) returns 0 if **nsdispatch**() returns NS_SUCCESS or NS_NOTFOUND, and **retval* otherwise.

getgrgid

struct group ***retval*, *gid_t gid*

**retval* should be set to a pointer to an internal static *struct group* on success, NULL otherwise.

getgrgid(3) returns **retval* if **nsdispatch**() returns NS_SUCCESS, NULL otherwise.

getgrgid_r
*int *retval*, *gid_t gid*, *struct group *grp*, *char *buffer*, *size_t buflen*, *struct group **result*

**retval* should be set to an appropriate errno(2) on failure.

getgrgid_r(3) returns 0 if **nsdispatch**() returns NS_SUCCESS or NS_NOTFOUND, and **retval* otherwise.

getgrnam
*struct group **retval*, *const char *name*

**retval* should be set to a pointer to an internal static *struct group* on success, NULL otherwise.

getgrnam(3) returns **retval* if **nsdispatch**() returns NS_SUCCESS, NULL otherwise.

getgrnam_r
*int *retval*, *const char *name*, *struct group *grp*, *char *buffer*, *size_t buflen*, *struct group **result*

**retval* should be set to an appropriate errno(2) on failure.

getgrnam_r(3) returns 0 if **nsdispatch**() returns NS_SUCCESS or NS_NOTFOUND, and **retval* otherwise.

getgroupmembership
*int *retval*, *const char *name*, *gid_t basegid*, *gid_t *groups*, *int maxgrp*, *int *groupc*

retval is unused.

Lookups for **group_compat** are also stopped if NS_SUCCESS was returned to prevent multiple "+:" compat entries from being expanded.

getgroupmembership(3) returns is -1 if **groupc* is greater than to *maxgrp*, and 0 otherwise.

setgroupent
*int *retval*, *int stayopen*

retval should be set to 0 on failure and 1 on success.

All methods for all sources are invoked for this method name.

setgrent
Empty *ap*.

All methods for all sources are invoked for this method name.

Methods for netgroup database
NOTE: The method APIs for this database will be changing in the near future.

endnetgrent
Empty *ap*.

lookup
*char *name, char **line, int bywhat*

Find the given *name* and return its value in *line*. *bywhat* is one of _NG_KEYBYNAME, _NG_KEYBYUSER, or _NG_KEYBYHOST.

getnetgrent
*int *retval, const char **host, const char **user, const char **domain*

**retval* should be set to 0 for no more netgroup members and 1 otherwise.

getnetgrent(3) returns **retval* if **nsdispatch**() returns NS_SUCCESS, 0 otherwise.

innetgr
*int *retval, const char *grp, const char *host, const char *user, const char *domain*

**retval* should be set to 1 for a successful match and 0 otherwise.

setnetgrent
*const char *netgroup*

Methods for networks database

getnetbyaddr
*struct netent **retval, uint32_t net, int type*

**retval* should be set to a pointer to an internal static *struct netent* on success, NULL otherwise.

getnetbyaddr(3) returns **retval* if **nsdispatch**() returns NS_SUCCESS, NULL otherwise.

getnetbyname
*struct netent **retval, const char *name*

**retval* should be set to a pointer to an internal static *struct netent* on success, NULL otherwise.

getnetbyname(3) returns **retval* if **nsdispatch**() returns NS_SUCCESS, NULL otherwise.

Methods for passwd and passwd_compat databases

endpwent
Empty *ap*.

All methods for all sources are invoked for this method name.

getpwent
*struct passwd **retval*

**retval* should be set to a pointer to an internal static *struct passwd* on success, NULL otherwise.

getpwent(3) returns **retval* if **nsdispatch**() returns NS_SUCCESS, NULL otherwise.

getpwent_r
*int *retval, struct passwd *pw, char *buffer, size_t buflen, struct passwd **result*

**retval* should be set to an appropriate errno(2) on failure.

getpwent_r(3) returns 0 if **nsdispatch**() returns NS_SUCCESS or NS_NOTFOUND, and **retval* otherwise.

getpwnam
*struct passwd **retval*, *const char *name*

**retval* should be set to a pointer to an internal static *struct passwd* on success, NULL otherwise.

getpwnam(3) returns **retval* if **nsdispatch**() returns NS_SUCCESS, NULL otherwise.

getpwnam_r
*int *retval*, *const char *name*, *struct passwd *pw*, *char *buffer*, *size_t buflen*, *struct passwd **result*

**retval* should be set to an appropriate errno(2) on failure.

getpwnam_r(3) returns 0 if **nsdispatch**() returns NS_SUCCESS or NS_NOTFOUND, and **retval* otherwise.

getpwuid
*struct passwd **retval*, *uid_t uid*

**retval* should be set to a pointer to an internal static *struct passwd* on success, NULL otherwise.

getpwuid(3) returns **retval* if **nsdispatch**() returns NS_SUCCESS, NULL otherwise.

getpwuid_r
*int *retval*, *uid_t uid*, *struct passwd *pw*, *char *buffer*, *size_t buflen*, *struct passwd **result*

**retval* should be set to an appropriate errno(2) on failure.

getpwuid_r returns 0 if **nsdispatch**() returns NS_SUCCESS or NS_NOTFOUND, and **retval* otherwise.

setpassent
*int *retval*, *int stayopen*

retval should be set to 0 on failure and 1 on success.

All methods for all sources are invoked for this method name.

setpwent
Empty *ap*.

All methods for all sources are invoked for this method name.

Methods for shells database

endusershell
Empty *ap*.

All methods for all sources are invoked for this method name.

getusershell
*char **retval*

getusershell(3) returns **retval* if **nsdispatch**() returns NS_SUCCESS, and 0 otherwise.

setusershell
Empty *ap*.

All methods for all sources are invoked for this method name.

SEE ALSO

`ld.elf_so`(1), `hesiod`(3), `stdarg`(3), `ypclnt`(3), `nsswitch.conf`(5)

HISTORY

The **`nsdispatch`** routines first appeared in NetBSD 1.4. Support for dynamically-loaded modules first appeared in NetBSD 3.0.

AUTHORS

Luke Mewburn ⟨lukem@NetBSD.org⟩ wrote this freely distributable name-service switch implementation, using ideas from the ULTRIX `svc.conf`(5) and Solaris `nsswitch.conf`(4) manual pages. Support for dynamically-loaded modules was added by Jason Thorpe ⟨thorpej@NetBSD.org⟩, based on code developed by the FreeBSD Project.

NAME

offtime, **timeoff**, **timegm**, **timelocal** — convert date and time

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <time.h>

struct tm *
offtime(const time_t * clock, long int offset);

time_t
timeoff(struct tm * tm, long int offset);

time_t
timegm(struct tm * tm);

time_t
timelocal(struct tm * tm);
```

DESCRIPTION

These functions are inspired by C standard interfaces named similarly.

offtime() converts the calendar time *clock*, offset by *offset* seconds, into broken-down time, expressed as Coordinated Universal Time (UTC).

timeoff() converts the broken-down time *tm*, expressed as UTC, offset by *offset* seconds, into a calendar time value.

timegm() converts the broken-down time *tm* into a calendar time value, effectively being the inverse of gmtime(3). It is equivalent to the C standard function mktime(3) operating in UTC.

timelocal() converts the broken down time *tm*, expressed as local time, into a calendar time value. It is equivalent to the C standard function mktime(3), and is provided for symmetry only.

SEE ALSO

ctime(3), tzset(3)

NAME

OMAPI - Object Management Application Programming Interface

DESCRIPTION

OMAPI is an programming layer designed for controlling remote applications, and for querying them for their state. It is currently used by the ISC DHCP server and this outline addresses the parts of OMAPI appropriate to the clients of DHCP server. It does this by also describing the use of a thin API layered on top of OMAPI called

OMAPI uses TCP/IP as the transport for server communication, and security can be imposed by having the client and server cryptographically sign messages using a shared secret.

dhcpctl works by presenting the client with handles to objects that act as surrogates for the real objects in the server. For example a client will create a handle for a lease object, and will request the server to fill the lease handle's state. The client application can then pull details such as the lease expiration time from the lease handle.

Modifications can be made to the server state by creating handles to new objects, or by modifying attributes of handles to existing objects, and then instructing the server to update itself according to the changes made.

USAGE

The client application must always call dhcpctl_initialize() before making calls to any other dhcpctl functions. This initializes various internal data structures.

To create the connection to the server the client must use dhcpctl_connect() function. As well as making the physical connection it will also set up the connection data structures to do authentication on each message, if that is required.

All the dhcpctl functions return an integer value of type isc_result_t. A successful call will yield a result of ISC_R_SUCCESS. If the call fails for a reason local to the client (e.g. insufficient local memory, or invalid arguments to the call) then the return value of the dhcpctl function will show that. If the call succeeds but the server couldn't process the request the error value from the server is returned through another way, shown below.

The easiest way to understand dhcpctl is to see it in action. The following program is fully functional, but almost all error checking has been removed to make is shorter and easier to understand. This program will query the server running on the localhost for the details of the lease for IP address 10.0.0.101. It will then print out the time the lease ends.

```
#include <stdarg.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>

#include <isc/result.h>
#include <dhcpctl/dhcpctl.h>

int main (int argc, char **argv) {
        dhcpctl_data_string ipaddrstring = NULL;
        dhcpctl_data_string value = NULL;
```

All modifications of handles and all accesses of handle data happen via dhcpctl_data_string objects.

```
        dhcpctl_handle connection = NULL;
        dhcpctl_handle lease = NULL;
        isc_result_t waitstatus;
        struct in_addr convaddr;
        time_t thetime;
```

```
dhcpctl_initialize ();
```

Required first step.

```
dhcpctl_connect (&connection, "127.0.0.1",
                 7911, 0);
```

Sets up the connection to the server. The server normally listens on port 7911 unless configured to do otherwise.

```
dhcpctl_new_object (&lease, connection,
                    "lease");
```

Here we create a handle to a lease. This call just sets up local data structure. The server hasn't yet made any association between the client's data structure and any lease it has.

```
memset (&ipaddrstring, 0, sizeof
        ipaddrstring);

inet_pton(AF_INET, "10.0.0.101",
          &convaddr);

omapi_data_string_new (&ipaddrstring,
                       4, MDL);
```

Create a new data string to storing in the handle.

```
memcpy(ipaddrstring->value, &convaddr.s_addr, 4);

dhcpctl_set_value (lease, ipaddrstring,
                   "ip-address");
```

We're setting the ip-address attribute of the lease handle to the given address. We've not set any other attributes so when the server makes the association the ip address will be all it uses to look up the lease in its tables.

```
dhcpctl_open_object (lease, connection, 0);
```

Here we prime the connection with the request to look up the lease in the server and fill up the local handle with the attributes the server will send over in its answer.

```
dhcpctl_wait_for_completion (lease,
                             &waitstatus);
```

This call causes the message to get sent to the server (the message to look up the lease and send back the attribute values in the answer). The value in the variable waitstatus when the function returns will be the result from the server. If the message could not be processed properly by the server then the error will be reflected here.

```
if (waitstatus != ISC_R_SUCCESS) {
        /* server not authoritative */
        exit (0);
}

dhcpctl_data_string_dereference(&ipaddrstring,
                                MDL);
```

Clean-up memory we no longer need.

```
dhcpctl_get_value (&value, lease, "ends");
```

Get the attribute named ‘‘ends’’ from the lease handle. This is a 4-byte integer of the time (in unix epoch seconds) that the lease will expire.

```
        memcpy(&thetime, value->value, value->len);
        dhcpctl_data_string_dereference(&value, MDL);

        fprintf (stdout, "ending time is %s",
                 ctime(&thetime));
}
```

AUTHENTICATION

If the server demands authenticated connections then before opening the connection the user must call dhcpctl_new_authenticator.

```
dhcpctl_handle authenticator = NULL;
const char *keyname = "a-key-name";
const char *algorithm = "hmac-md5";
const char *secret = "a-shared-secret";

dhcpctl_new_authenticator (&authenticator,
        keyname,
        algorithm,
        secret,
                        strlen(secret) + 1);
```

The keyname, algorithm and must all match what is specified in the server's dhcpd.conf file, excepting that the secret should appear in 'raw' form, not in base64 as it would in dhcpd.conf:

```
key "a-key-name" {
        algorithm hmac-md5;
        secret "a-shared-secret";
};

# Set the omapi-key value to use
# authenticated connections
omapi-key a-key-name;
```

The authenticator handle that is created by the call to dhcpctl_new_authenticator must be given as the last (the 4th) argument to the call to dhcpctl_connect(). All messages will then be signed with the given secret string using the specified algorithm.

SEE ALSO

dhcpctl(3), omapi(3), dhcpd(8), dhclient(8), dhcpd.conf(5), dhclient.conf(5).

AUTHOR

omapi was created by Ted Lemon of Nominum, Inc. Information about Nominum and support contracts for DHCP and BIND can be found at **http://www.nominum.com. This documentation was written by James** Brister of Nominum, Inc.

NAME

opendisk — open a disk partition

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
opendisk(const char *path, int flags, char *buf, size_t buflen, int iscooked);
```

DESCRIPTION

opendisk() opens *path*, for reading and/or writing as specified by the argument *flags* using open(2), and the file descriptor is returned to the caller. *buf* is used to store the resultant filename. *buflen* is the size, in bytes, of the array referenced by *buf* (usually MAXPATHLEN bytes). *iscooked* controls which paths in /dev are tried.

opendisk() attempts to open the following variations of *path*, in order:

path — The pathname as given.

path*X* — *path* with a suffix of '*X*', where '*X*' represents the raw partition of the device, as determined by getrawpartition(3), usually "c".

If *path* does not contain a slash ("/"), the following variations are attempted:

- If *iscooked* is zero:

  /dev/rpath — *path* with a prefix of "/dev/r".

  /dev/rpath*X* — *path* with a prefix of "/dev/r" and a suffix of '*X*' (q.v.).

- If *iscooked* is non-zero:

  /dev/path — *path* with a prefix of "/dev/".

  /dev/path*X* — *path* with a prefix of "/dev/" and a suffix of '*X*' (q.v.).

RETURN VALUES

An open file descriptor, or -1 if the open(2) failed.

ERRORS

opendisk() may set *errno* to one of the following values:

[EINVAL] — O_CREAT was set in *flags*, or getrawpartition(3) didn't return a valid partition.

[EFAULT] — *buf* was the NULL pointer.

The **opendisk**() function may also set *errno* to any value specified by the open(2) function.

SEE ALSO

open(2), getrawpartition(3)

HISTORY

The **opendisk**() function first appeared in NetBSD 1.3.

NAME

openpam_borrow_cred, **openpam_free_data**, **openpam_free_envlist**, **openpam_get_option**, **openpam_log**, **openpam_nullconv**, **openpam_readline**, **openpam_restore_cred**, **openpam_set_option**, **openpam_ttyconv**, **pam_error**, **pam_get_authtok**, **pam_info**, **pam_prompt**, **pam_setenv**, **pam_verror**, **pam_vinfo**, **pam_vprompt** — Pluggable Authentication Modules Library

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <security/openpam.h>

int
openpam_borrow_cred(pam_handle_t *pamh, const struct passwd *pwd);

void
openpam_free_data(pam_handle_t *pamh, void *data, int status);

void
openpam_free_envlist(char **envlist);

const char *
openpam_get_option(pam_handle_t *pamh, const char *option);

void
openpam_log(int level, const char *fmt, ...);

int
openpam_nullconv(int n, const struct pam_message **msg,
    struct pam_response **resp, void *data);

char *
openpam_readline(FILE *f, int *lineno, size_t *lenp);

int
openpam_restore_cred(pam_handle_t *pamh);

int
openpam_set_option(pam_handle_t *pamh, const char *option,
    const char *value);

int
openpam_ttyconv(int n, const struct pam_message **msg,
    struct pam_response **resp, void *data);

int
pam_error(const pam_handle_t *pamh, const char *fmt, ...);

int
pam_get_authtok(pam_handle_t *pamh, int item, const char **authtok,
    const char *prompt);

int
pam_info(const pam_handle_t *pamh, const char *fmt, ...);

int
pam_prompt(const pam_handle_t *pamh, int style, char **resp,
    const char *fmt, ...);
```

```
int
pam_setenv(pam_handle_t *pamh, const char *name, const char *value,
    int overwrite);

int
pam_verror(const pam_handle_t *pamh, const char *fmt, va_list ap);

int
pam_vinfo(const pam_handle_t *pamh, const char *fmt, va_list ap);

int
pam_vprompt(const pam_handle_t *pamh, int style, char **resp,
    const char *fmt, va_list ap);
```

DESCRIPTION

These functions are OpenPAM extensions to the PAM API. Those named **pam**_*() are, in the author's opinion, logical and necessary extensions to the standard API, while those named **openpam**_*() are either simple convenience functions, or functions intimately tied to OpenPAM implementation details, and therefore not well suited to standardization.

SEE ALSO

openpam_borrow_cred(3), openpam_free_data(3), openpam_free_envlist(3), openpam_get_option(3), openpam_log(3), openpam_nullconv(3), openpam_readline(3), openpam_restore_cred(3), openpam_set_option(3), openpam_ttyconv(3), pam_error(3), pam_get_authtok(3), pam_info(3), pam_prompt(3), pam_setenv(3), pam_verror(3), pam_vinfo(3), pam_vprompt(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The OpenPAM library and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

openpam_borrow_cred — temporarily borrow user credentials

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/openpam.h>

int
openpam_borrow_cred(pam_handle_t *pamh, const struct passwd *pwd);
```

DESCRIPTION

The **openpam_borrow_cred** function saves the current credentials and switches to those of the user specified by its *pwd* argument. The affected credentials are the effective UID, the effective GID, and the group access list. The original credentials can be restored using openpam_restore_cred(3).

RETURN VALUES

The **openpam_borrow_cred** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_PERM_DENIED] Permission denied.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

setegid(2), seteuid(2), setgroups(2), openpam_restore_cred(3), pam(3), pam_strerror(3)

STANDARDS

The **openpam_borrow_cred** function is an OpenPAM extension.

AUTHORS

The **openpam_borrow_cred** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

openpam_free_data — generic cleanup function

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/openpam.h>

void
openpam_free_data(pam_handle_t *pamh, void *data, int status);
```

DESCRIPTION

The **openpam_free_data** function is a cleanup function suitable for passing to pam_set_data(3). It simply releases the data by passing its *data* argument to free(3).

SEE ALSO

free(3), pam(3), pam_set_data(3)

STANDARDS

The **openpam_free_data** function is an OpenPAM extension.

AUTHORS

The **openpam_free_data** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

openpam_free_envlist — free an environment list

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/openpam.h>

void
openpam_free_envlist(char **envlist);
```

DESCRIPTION

The **openpam_free_envlist** function is a convenience function which frees all the environment variables in an environment list, and the list itself. It is suitable for freeing the return value from pam_getenvlist(3).

SEE ALSO

pam(3), pam_getenvlist(3)

STANDARDS

The **openpam_free_envlist** function is an OpenPAM extension.

AUTHORS

The **openpam_free_envlist** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

openpam_get_option — returns the value of a module option

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/openpam.h>

const char *
openpam_get_option(pam_handle_t *pamh, const char *option);
```

DESCRIPTION

The **openpam_get_option** function returns the value of the specified option in the context of the currently executing service module, or NULL if the option is not set or no module is currently executing.

RETURN VALUES

The **openpam_get_option** function returns NULL on failure.

SEE ALSO

openpam_set_option(3), pam(3)

STANDARDS

The **openpam_get_option** function is an OpenPAM extension.

AUTHORS

The **openpam_get_option** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

openpam_log — log a message through syslog

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/openpam.h>

void
openpam_log(int level, const char *fmt, ...);
```

DESCRIPTION

The **openpam_log** function logs messages using `syslog`(3). It is primarily intended for internal use by the library and modules.

The *level* argument indicates the importance of the message. The following levels are defined:

`PAM_LOG_DEBUG` Debugging messages. These messages are normally not logged unless the global integer variable *_openpam_debug* is set to a non-zero value, in which case they are logged with a `syslog`(3) priority of `LOG_DEBUG`.

`PAM_LOG_VERBOSE` Information about the progress of the authentication process, or other non-essential messages. These messages are logged with a `syslog`(3) priority of `LOG_INFO`.

`PAM_LOG_NOTICE` Messages relating to non-fatal errors. These messages are logged with a `syslog`(3) priority of `LOG_NOTICE`.

`PAM_LOG_ERROR` Messages relating to serious errors. These messages are logged with a `syslog`(3) priority of `LOG_ERR`.

The remaining arguments are a `printf`(3) format string and the corresponding arguments.

SEE ALSO

`pam`(3), `printf`(3), `syslog`(3)

STANDARDS

The **openpam_log** function is an OpenPAM extension.

AUTHORS

The **openpam_log** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

openpam_nullconv — null conversation function

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/openpam.h>

int
openpam_nullconv(int n, const struct pam_message **msg,
    struct pam_response **resp, void *data);
```

DESCRIPTION

The **openpam_nullconv** function is a null conversation function suitable for applications that want to use PAM but don't support interactive dialog with the user. Such applications should set PAM_AUTHTOK to whatever authentication token they've obtained on their own before calling pam_authenticate(3) and / or pam_chauthtok(3), and their PAM configuration should specify the use_first_pass option for all modules that require access to the authentication token, to make sure they use PAM_AUTHTOK rather than try to query the user.

RETURN VALUES

The **openpam_nullconv** function returns one of the following values:

[PAM_CONV_ERR] Conversation failure.

SEE ALSO

openpam_ttyconv(3), pam(3), pam_authenticate(3), pam_chauthtok(3), pam_prompt(3), pam_set_item(3), pam_strerror(3), pam_vprompt(3)

STANDARDS

The **openpam_nullconv** function is an OpenPAM extension.

AUTHORS

The **openpam_nullconv** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

openpam_readline — read a line from a file

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/openpam.h>

char *
openpam_readline(FILE *f, int *lineno, size_t *lenp);
```

DESCRIPTION

The **openpam_readline** function reads a line from a file, and returns it in a NUL-terminated buffer allocated with malloc(3).

The **openpam_readline** function performs a certain amount of processing on the data it reads. Comments (introduced by a hash sign) are stripped, as is leading and trailing whitespace. Any amount of linear whitespace is collapsed to a single space. Blank lines are ignored. If a line ends in a backslash, the backslash is stripped and the next line is appended.

If *lineno* is not NULL, the integer variable it points to is incremented every time a newline character is read.

If *lenp* is not NULL, the length of the line (not including the terminating NUL character) is stored in the variable it points to.

The caller is responsible for releasing the returned buffer by passing it to free(3).

RETURN VALUES

The **openpam_readline** function returns NULL on failure.

SEE ALSO

free(3), malloc(3), pam(3)

STANDARDS

The **openpam_readline** function is an OpenPAM extension.

AUTHORS

The **openpam_readline** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

openpam_restore_cred — restore credentials

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/openpam.h>

int
openpam_restore_cred(pam_handle_t *pamh);
```

DESCRIPTION

The **openpam_restore_cred** function restores the credentials saved by openpam_borrow_cred(3).

RETURN VALUES

The **openpam_restore_cred** function returns one of the following values:

[PAM_NO_MODULE_DATA]
: Module data not found.

[PAM_SYSTEM_ERR]
: System error.

SEE ALSO

setegid(2), seteuid(2), setgroups(2), openpam_borrow_cred(3), pam(3), pam_strerror(3)

STANDARDS

The **openpam_restore_cred** function is an OpenPAM extension.

AUTHORS

The **openpam_restore_cred** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

openpam_set_option — sets the value of a module option

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/openpam.h>

int
openpam_set_option(pam_handle_t *pamh, const char *option,
    const char *value);
```

DESCRIPTION

The **openpam_set_option** function sets the specified option in the context of the currently executing service module.

RETURN VALUES

The **openpam_set_option** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

openpam_get_option(3), pam(3), pam_strerror(3)

STANDARDS

The **openpam_set_option** function is an OpenPAM extension.

AUTHORS

The **openpam_set_option** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

openpam_ttyconv — simple tty-based conversation function

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/openpam.h>

int
openpam_ttyconv(int n, const struct pam_message **msg,
    struct pam_response **resp, void *data);
```

DESCRIPTION

The **openpam_ttyconv** function is a standard conversation function suitable for use on TTY devices. It should be adequate for the needs of most text-based interactive programs.

The **openpam_ttyconv** function displays a prompt to, and reads in a password from /dev/tty. If this file is not accessible, **openpam_ttyconv** displays the prompt on the standard error output and reads from the standard input.

The **openpam_ttyconv** function allows the application to specify a timeout for user input by setting the global integer variable *openpam_ttyconv_timeout* to the length of the timeout in seconds.

RETURN VALUES

The **openpam_ttyconv** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

getpass(3), openpam_nullconv(3), pam(3), pam_prompt(3), pam_strerror(3), pam_vprompt(3)

STANDARDS

The **openpam_ttyconv** function is an OpenPAM extension.

AUTHORS

The **openpam_ttyconv** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

openpty, **login_tty**, **forkpty** — tty utility functions

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
openpty(int *amaster, int *aslave, char *name, struct termios *termp,
    struct winsize *winp);

int
login_tty(int fd);

pid_t
forkpty(int *amaster, char *name, struct termios *termp,
    struct winsize *winp);
```

DESCRIPTION

The **openpty**(), **login_tty**(), and **forkpty**() functions perform manipulations on ttys and pseudo-ttys.

The **openpty**() function finds an available pseudo-tty and returns file descriptors for the master and slave in *amaster* and *aslave*. If *name* is non-null, the filename of the slave is returned in *name*. If *termp* is non-null, the terminal parameters of the slave will be set to the values in *termp*. If *winp* is non-null, the window size of the slave will be set to the values in *winp*.

The **login_tty**() function prepares for a login on the tty *fd* (which may be a real tty device, or the slave of a pseudo-tty as returned by **openpty**()) by creating a new session, making *fd* the controlling terminal for the current process, setting *fd* to be the standard input, output, and error streams of the current process, and closing *fd*.

The **forkpty**() function combines **openpty**(), **fork**(), and **login_tty**() to create a new process operating in a pseudo-tty. The file descriptor of the master side of the pseudo-tty is returned in *amaster*, and the filename of the slave in *name* if it is non-null. The *termp* and *winp* parameters, if non-null, will determine the terminal attributes and window size of the slave side of the pseudo-tty.

RETURN VALUES

If a call to **openpty**(), **login_tty**(), or **forkpty**() is not successful, -1 is returned and *errno* is set to indicate the error. Otherwise, **openpty**(), **login_tty**(), and the child process of **forkpty**() return 0, and the parent process of **forkpty**() returns the process ID of the child process.

FILES

`/dev/[pt]ty[p-zP-T][0-9a-zA-Z]`

ERRORS

openpty() will fail if:

`[ENOENT]` There are no available ttys.

`[EPERM]` The caller was not the superuser and the `ptm`(4) device is missing or not configured.

login_tty() will fail if **ioctl**() fails to set *fd* to the controlling terminal of the current process. **forkpty**() will fail if either **openpty**() or **fork**() fails.

SEE ALSO

`fork`(2)

NAME

bio − I/O abstraction

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bio.h>
```

TBA

DESCRIPTION

A BIO is an I/O abstraction, it hides many of the underlying I/O details from an application. If an application uses a BIO for its I/O it can transparently handle SSL connections, unencrypted network connections and file I/O.

There are two type of BIO, a source/sink BIO and a filter BIO.

As its name implies a source/sink BIO is a source and/or sink of data, examples include a socket BIO and a file BIO.

A filter BIO takes data from one BIO and passes it through to another, or the application. The data may be left unmodified (for example a message digest BIO) or translated (for example an encryption BIO). The effect of a filter BIO may change according to the I/O operation it is performing: for example an encryption BIO will encrypt data if it is being written to and decrypt data if it is being read from.

BIOs can be joined together to form a chain (a single BIO is a chain with one component). A chain normally consist of one source/sink BIO and one or more filter BIOs. Data read from or written to the first BIO then traverses the chain to the end (normally a source/sink BIO).

SEE ALSO

BIO_ctrl (3), *BIO_f_base64* (3), *BIO_f_buffer* (3), *BIO_f_cipher* (3), *BIO_f_md* (3), *BIO_f_null* (3), *BIO_f_ssl* (3), *BIO_find_type* (3), *BIO_new* (3), *BIO_new_bio_pair* (3), *BIO_push* (3), *BIO_read* (3), *BIO_s_accept* (3), *BIO_s_bio* (3), *BIO_s_connect* (3), *BIO_s_fd* (3), *BIO_s_file* (3), *BIO_s_mem* (3), *BIO_s_null* (3), *BIO_s_socket* (3), *BIO_set_callback* (3), *BIO_should_retry* (3)

NAME

blowfish, BF_set_key, BF_encrypt, BF_decrypt, BF_ecb_encrypt, BF_cbc_encrypt, BF_cfb64_encrypt, BF_ofb64_encrypt, BF_options − Blowfish encryption

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/blowfish.h>

void BF_set_key(BF_KEY *key, int len, const unsigned char *data);

void BF_ecb_encrypt(const unsigned char *in, unsigned char *out,
        BF_KEY *key, int enc);
void BF_cbc_encrypt(const unsigned char *in, unsigned char *out,
        long length, BF_KEY *schedule, unsigned char *ivec, int enc);
void BF_cfb64_encrypt(const unsigned char *in, unsigned char *out,
        long length, BF_KEY *schedule, unsigned char *ivec, int *num,
        int enc);
void BF_ofb64_encrypt(const unsigned char *in, unsigned char *out,
        long length, BF_KEY *schedule, unsigned char *ivec, int *num);
const char *BF_options(void);

void BF_encrypt(BF_LONG *data,const BF_KEY *key);
void BF_decrypt(BF_LONG *data,const BF_KEY *key);
```

DESCRIPTION

This library implements the Blowfish cipher, which was invented and described by Counterpane (see http://www.counterpane.com/blowfish.html).

Blowfish is a block cipher that operates on 64 bit (8 byte) blocks of data. It uses a variable size key, but typically, 128 bit (16 byte) keys are considered good for strong encryption. Blowfish can be used in the same modes as DES (see *des_modes*(7)). Blowfish is currently one of the faster block ciphers. It is quite a bit faster than DES, and much faster than IDEA or RC2.

Blowfish consists of a key setup phase and the actual encryption or decryption phase.

BF_set_key() sets up the **BF_KEY key** using the **len** bytes long key at **data**.

BF_ecb_encrypt() is the basic Blowfish encryption and decryption function. It encrypts or decrypts the first 64 bits of **in** using the key **key**, putting the result in **out**. **enc** decides if encryption (**BF_ENCRYPT**) or decryption (**BF_DECRYPT**) shall be performed. The vector pointed at by **in** and **out** must be 64 bits in length, no less. If they are larger, everything after the first 64 bits is ignored.

The mode functions *BF_cbc_encrypt()*, *BF_cfb64_encrypt()* and *BF_ofb64_encrypt()* all operate on variable length data. They all take an initialization vector **ivec** which needs to be passed along into the next call of the same function for the same message. **ivec** may be initialized with anything, but the recipient needs to know what it was initialized with, or it won't be able to decrypt. Some programs and protocols simplify this, like SSH, where **ivec** is simply initialized to zero. *BF_cbc_encrypt()* operates on data that is a multiple of 8 bytes long, while *BF_cfb64_encrypt()* and *BF_ofb64_encrypt()* are used to encrypt an variable number of bytes (the amount does not have to be an exact multiple of 8). The purpose of the latter two is to simulate stream ciphers, and therefore, they need the parameter **num**, which is a pointer to an integer where the current offset in **ivec** is stored between calls. This integer must be initialized to zero when **ivec** is initialized.

BF_cbc_encrypt() is the Cipher Block Chaining function for Blowfish. It encrypts or decrypts the 64 bits chunks of **in** using the key **schedule**, putting the result in **out**. **enc** decides if encryption (BF_ENCRYPT) or decryption (BF_DECRYPT) shall be performed. **ivec** must point at an 8 byte long initialization vector.

BF_cfb64_encrypt() is the CFB mode for Blowfish with 64 bit feedback. It encrypts or decrypts the bytes in **in** using the key **schedule**, putting the result in **out**. **enc** decides if encryption (**BF_ENCRYPT**) or decryption (**BF_DECRYPT**) shall be performed. **ivec** must point at an 8 byte long initialization vector. **num** must

point at an integer which must be initially zero.

BF_ofb64_encrypt() is the OFB mode for Blowfish with 64 bit feedback. It uses the same parameters as *BF_cfb64_encrypt()*, which must be initialized the same way.

BF_encrypt() and *BF_decrypt()* are the lowest level functions for Blowfish encryption. They encrypt/decrypt the first 64 bits of the vector pointed by **data**, using the key **key**. These functions should not be used unless you implement 'modes' of Blowfish. The alternative is to use *BF_ecb_encrypt()*. If you still want to use these functions, you should be aware that they take each 32−bit chunk in host-byte order, which is little-endian on little-endian platforms and big-endian on big-endian ones.

RETURN VALUES

None of the functions presented here return any value.

NOTE

Applications should use the higher level functions *EVP_EncryptInit*(3) etc. instead of calling the blowfish functions directly.

SEE ALSO

des_modes(7)

HISTORY

The Blowfish functions are available in all versions of SSLeay and OpenSSL.

NAME

bn − multiprecision integer arithmetics

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

BIGNUM *BN_new(void);
void BN_free(BIGNUM *a);
void BN_init(BIGNUM *);
void BN_clear(BIGNUM *a);
void BN_clear_free(BIGNUM *a);

BN_CTX *BN_CTX_new(void);
void BN_CTX_init(BN_CTX *c);
void BN_CTX_free(BN_CTX *c);

BIGNUM *BN_copy(BIGNUM *a, const BIGNUM *b);
BIGNUM *BN_dup(const BIGNUM *a);

BIGNUM *BN_swap(BIGNUM *a, BIGNUM *b);

int BN_num_bytes(const BIGNUM *a);
int BN_num_bits(const BIGNUM *a);
int BN_num_bits_word(BN_ULONG w);

void BN_set_negative(BIGNUM *a, int n);
int  BN_is_negative(const BIGNUM *a);

int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *b);
int BN_sub(BIGNUM *r, const BIGNUM *a, const BIGNUM *b);
int BN_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
int BN_sqr(BIGNUM *r, BIGNUM *a, BN_CTX *ctx);
int BN_div(BIGNUM *dv, BIGNUM *rem, const BIGNUM *a, const BIGNUM *d,
        BN_CTX *ctx);
int BN_mod(BIGNUM *rem, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
int BN_nnmod(BIGNUM *rem, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_add(BIGNUM *ret, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);
int BN_mod_sub(BIGNUM *ret, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);
int BN_mod_mul(BIGNUM *ret, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);
int BN_mod_sqr(BIGNUM *ret, BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
int BN_exp(BIGNUM *r, BIGNUM *a, BIGNUM *p, BN_CTX *ctx);
int BN_mod_exp(BIGNUM *r, BIGNUM *a, const BIGNUM *p,
        const BIGNUM *m, BN_CTX *ctx);
int BN_gcd(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);

int BN_add_word(BIGNUM *a, BN_ULONG w);
int BN_sub_word(BIGNUM *a, BN_ULONG w);
int BN_mul_word(BIGNUM *a, BN_ULONG w);
BN_ULONG BN_div_word(BIGNUM *a, BN_ULONG w);
BN_ULONG BN_mod_word(const BIGNUM *a, BN_ULONG w);
```

```
int BN_cmp(BIGNUM *a, BIGNUM *b);
int BN_ucmp(BIGNUM *a, BIGNUM *b);
int BN_is_zero(BIGNUM *a);
int BN_is_one(BIGNUM *a);
int BN_is_word(BIGNUM *a, BN_ULONG w);
int BN_is_odd(BIGNUM *a);

int BN_zero(BIGNUM *a);
int BN_one(BIGNUM *a);
const BIGNUM *BN_value_one(void);
int BN_set_word(BIGNUM *a, unsigned long w);
unsigned long BN_get_word(BIGNUM *a);

int BN_rand(BIGNUM *rnd, int bits, int top, int bottom);
int BN_pseudo_rand(BIGNUM *rnd, int bits, int top, int bottom);
int BN_rand_range(BIGNUM *rnd, BIGNUM *range);
int BN_pseudo_rand_range(BIGNUM *rnd, BIGNUM *range);

BIGNUM *BN_generate_prime(BIGNUM *ret, int bits,int safe, BIGNUM *add,
        BIGNUM *rem, void (*callback)(int, int, void *), void *cb_arg);
int BN_is_prime(const BIGNUM *p, int nchecks,
        void (*callback)(int, int, void *), BN_CTX *ctx, void *cb_arg);

int BN_set_bit(BIGNUM *a, int n);
int BN_clear_bit(BIGNUM *a, int n);
int BN_is_bit_set(const BIGNUM *a, int n);
int BN_mask_bits(BIGNUM *a, int n);
int BN_lshift(BIGNUM *r, const BIGNUM *a, int n);
int BN_lshift1(BIGNUM *r, BIGNUM *a);
int BN_rshift(BIGNUM *r, BIGNUM *a, int n);
int BN_rshift1(BIGNUM *r, BIGNUM *a);

int BN_bn2bin(const BIGNUM *a, unsigned char *to);
BIGNUM *BN_bin2bn(const unsigned char *s, int len, BIGNUM *ret);
char *BN_bn2hex(const BIGNUM *a);
char *BN_bn2dec(const BIGNUM *a);
int BN_hex2bn(BIGNUM **a, const char *str);
int BN_dec2bn(BIGNUM **a, const char *str);
int BN_print(BIO *fp, const BIGNUM *a);
int BN_print_fp(FILE *fp, const BIGNUM *a);
int BN_bn2mpi(const BIGNUM *a, unsigned char *to);
BIGNUM *BN_mpi2bn(unsigned char *s, int len, BIGNUM *ret);

BIGNUM *BN_mod_inverse(BIGNUM *r, BIGNUM *a, const BIGNUM *n,
    BN_CTX *ctx);

BN_RECP_CTX *BN_RECP_CTX_new(void);
void BN_RECP_CTX_init(BN_RECP_CTX *recp);
void BN_RECP_CTX_free(BN_RECP_CTX *recp);
int BN_RECP_CTX_set(BN_RECP_CTX *recp, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_mul_reciprocal(BIGNUM *r, BIGNUM *a, BIGNUM *b,
       BN_RECP_CTX *recp, BN_CTX *ctx);
```

```
BN_MONT_CTX *BN_MONT_CTX_new(void);
void BN_MONT_CTX_init(BN_MONT_CTX *ctx);
void BN_MONT_CTX_free(BN_MONT_CTX *mont);
int BN_MONT_CTX_set(BN_MONT_CTX *mont, const BIGNUM *m, BN_CTX *ctx);
BN_MONT_CTX *BN_MONT_CTX_copy(BN_MONT_CTX *to, BN_MONT_CTX *from);
int BN_mod_mul_montgomery(BIGNUM *r, BIGNUM *a, BIGNUM *b,
        BN_MONT_CTX *mont, BN_CTX *ctx);
int BN_from_montgomery(BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont,
        BN_CTX *ctx);
int BN_to_montgomery(BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont,
        BN_CTX *ctx);

BN_BLINDING *BN_BLINDING_new(const BIGNUM *A, const BIGNUM *Ai,
       BIGNUM *mod);
void BN_BLINDING_free(BN_BLINDING *b);
int BN_BLINDING_update(BN_BLINDING *b,BN_CTX *ctx);
int BN_BLINDING_convert(BIGNUM *n, BN_BLINDING *b, BN_CTX *ctx);
int BN_BLINDING_invert(BIGNUM *n, BN_BLINDING *b, BN_CTX *ctx);
int BN_BLINDING_convert_ex(BIGNUM *n, BIGNUM *r, BN_BLINDING *b,
       BN_CTX *ctx);
int BN_BLINDING_invert_ex(BIGNUM *n,const BIGNUM *r,BN_BLINDING *b,
       BN_CTX *ctx);
void BN_BLINDING_set_thread(BN_BLINDING *);
int BN_BLINDING_cmp_thread(const BN_BLINDING *,
       const CRYPTO_THREADID *);

unsigned long BN_BLINDING_get_flags(const BN_BLINDING *);
void BN_BLINDING_set_flags(BN_BLINDING *, unsigned long);
BN_BLINDING *BN_BLINDING_create_param(BN_BLINDING *b,
       const BIGNUM *e, BIGNUM *m, BN_CTX *ctx,
       int (*bn_mod_exp)(BIGNUM *r, const BIGNUM *a, const BIGNUM *p,
                         const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *m_ctx),
       BN_MONT_CTX *m_ctx);
```

DESCRIPTION

This library performs arithmetic operations on integers of arbitrary size. It was written for use in public key cryptography, such as RSA and Diffie−Hellman.

It uses dynamic memory allocation for storing its data structures. That means that there is no limit on the size of the numbers manipulated by these functions, but return values must always be checked in case a memory allocation error has occurred.

The basic object in this library is a **BIGNUM**. It is used to hold a single large integer. This type should be considered opaque and fields should not be modified or accessed directly.

The creation of **BIGNUM** objects is described in *BN_new* (3); *BN_add* (3) describes most of the arithmetic operations. Comparison is described in *BN_cmp* (3); *BN_zero* (3) describes certain assignments, *BN_rand* (3) the generation of random numbers, *BN_generate_prime* (3) deals with prime numbers and *BN_set_bit* (3) with bit operations. The conversion of **BIGNUM**s to external formats is described in *BN_bn2bin* (3).

SEE ALSO

openssl_bn_internal (3), *openssl_dh* (3), *openssl_err* (3), *openssl_rand* (3), *openssl_rsa* (3), *BN_new* (3), *BN_CTX_new* (3), *BN_copy* (3), *BN_swap* (3), *BN_num_bytes* (3), *BN_add* (3), *BN_add_word* (3), *BN_cmp* (3), *BN_zero* (3), *BN_rand* (3), *BN_generate_prime* (3), *BN_set_bit* (3), *BN_bn2bin* (3), *BN_mod_inverse* (3), *BN_mod_mul_reciprocal* (3), *BN_mod_mul_montgomery* (3), *BN_BLINDING_new* (3)

NAME

bn_mul_words, bn_mul_add_words, bn_sqr_words, bn_div_words, bn_add_words, bn_sub_words, bn_mul_comba4, bn_mul_comba8, bn_sqr_comba4, bn_sqr_comba8, bn_cmp_words, bn_mul_normal, bn_mul_low_normal, bn_mul_recursive, bn_mul_part_recursive, bn_mul_low_recursive, bn_mul_high, bn_sqr_normal, bn_sqr_recursive, bn_expand, bn_wexpand, bn_expand2, bn_fix_top, bn_check_top, bn_print, bn_dump, bn_set_max, bn_set_high, bn_set_low − BIGNUM library internal functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/bn.h>

BN_ULONG bn_mul_words(BN_ULONG *rp, BN_ULONG *ap, int num, BN_ULONG w);
BN_ULONG bn_mul_add_words(BN_ULONG *rp, BN_ULONG *ap, int num,
  BN_ULONG w);
void     bn_sqr_words(BN_ULONG *rp, BN_ULONG *ap, int num);
BN_ULONG bn_div_words(BN_ULONG h, BN_ULONG l, BN_ULONG d);
BN_ULONG bn_add_words(BN_ULONG *rp, BN_ULONG *ap, BN_ULONG *bp,
  int num);
BN_ULONG bn_sub_words(BN_ULONG *rp, BN_ULONG *ap, BN_ULONG *bp,
  int num);

void bn_mul_comba4(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b);
void bn_mul_comba8(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b);
void bn_sqr_comba4(BN_ULONG *r, BN_ULONG *a);
void bn_sqr_comba8(BN_ULONG *r, BN_ULONG *a);

int bn_cmp_words(BN_ULONG *a, BN_ULONG *b, int n);

void bn_mul_normal(BN_ULONG *r, BN_ULONG *a, int na, BN_ULONG *b,
  int nb);
void bn_mul_low_normal(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n);
void bn_mul_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n2,
  int dna,int dnb,BN_ULONG *tmp);
void bn_mul_part_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b,
  int n, int tna,int tnb, BN_ULONG *tmp);
void bn_mul_low_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b,
  int n2, BN_ULONG *tmp);
void bn_mul_high(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, BN_ULONG *l,
  int n2, BN_ULONG *tmp);

void bn_sqr_normal(BN_ULONG *r, BN_ULONG *a, int n, BN_ULONG *tmp);
void bn_sqr_recursive(BN_ULONG *r, BN_ULONG *a, int n2, BN_ULONG *tmp);

void mul(BN_ULONG r, BN_ULONG a, BN_ULONG w, BN_ULONG c);
void mul_add(BN_ULONG r, BN_ULONG a, BN_ULONG w, BN_ULONG c);
void sqr(BN_ULONG r0, BN_ULONG r1, BN_ULONG a);

BIGNUM *bn_expand(BIGNUM *a, int bits);
BIGNUM *bn_wexpand(BIGNUM *a, int n);
BIGNUM *bn_expand2(BIGNUM *a, int n);
void bn_fix_top(BIGNUM *a);

void bn_check_top(BIGNUM *a);
void bn_print(BIGNUM *a);
void bn_dump(BN_ULONG *d, int n);
void bn_set_max(BIGNUM *a);
void bn_set_high(BIGNUM *r, BIGNUM *a, int n);
void bn_set_low(BIGNUM *r, BIGNUM *a, int n);
```

DESCRIPTION

This page documents the internal functions used by the OpenSSL **BIGNUM** implementation. They are described here to facilitate debugging and extending the library. They are *not* to be used by applications.

The BIGNUM structure

```
typedef struct bignum_st
       {
       int top;      /* number of words used in d */
       BN_ULONG *d;  /* pointer to an array containing the integer value */
       int max;      /* size of the d array */
       int neg;      /* sign */
       } BIGNUM;
```

The integer value is stored in **d**, a *malloc()*ed array of words (**BN_ULONG**), least significant word first. A **BN_ULONG** can be either 16, 32 or 64 bits in size, depending on the 'number of bits' (**BITS2**) specified in `openssl/bn.h`.

max is the size of the **d** array that has been allocated. **top** is the number of words being used, so for a value of 4, bn.d[0]=4 and bn.top=1. **neg** is 1 if the number is negative. When a **BIGNUM** is **0**, the **d** field can be **NULL** and **top** == **0**.

Various routines in this library require the use of temporary **BIGNUM** variables during their execution. Since dynamic memory allocation to create **BIGNUM**s is rather expensive when used in conjunction with repeated subroutine calls, the **BN_CTX** structure is used. This structure contains **BN_CTX_NUM** **BIGNUM**s, see *BN_CTX_start* (3).

Low-level arithmetic operations

These functions are implemented in C and for several platforms in assembly language:

bn_mul_words(**rp**, **ap**, **num**, **w**) operates on the **num** word arrays **rp** and **ap**. It computes **ap** * **w**, places the result in **rp**, and returns the high word (carry).

bn_mul_add_words(**rp**, **ap**, **num**, **w**) operates on the **num** word arrays **rp** and **ap**. It computes **ap** * **w** + **rp**, places the result in **rp**, and returns the high word (carry).

bn_sqr_words(**rp**, **ap**, **n**) operates on the **num** word array **ap** and the 2***num** word array **ap**. It computes **ap** * **ap** word−wise, and places the low and high bytes of the result in **rp**.

bn_div_words(**h**, **l**, **d**) divides the two word number (**h**,**l**) by **d** and returns the result.

bn_add_words(**rp**, **ap**, **bp**, **num**) operates on the **num** word arrays **ap**, **bp** and **rp**. It computes **ap** + **bp**, places the result in **rp**, and returns the high word (carry).

bn_sub_words(**rp**, **ap**, **bp**, **num**) operates on the **num** word arrays **ap**, **bp** and **rp**. It computes **ap** − **bp**, places the result in **rp**, and returns the carry (1 if **bp** > **ap**, 0 otherwise).

bn_mul_comba4(**r**, **a**, **b**) operates on the 4 word arrays **a** and **b** and the 8 word array **r**. It computes **a***b and places the result in **r**.

bn_mul_comba8(**r**, **a**, **b**) operates on the 8 word arrays **a** and **b** and the 16 word array **r**. It computes **a***b and places the result in **r**.

bn_sqr_comba4(**r**, **a**, **b**) operates on the 4 word arrays **a** and **b** and the 8 word array **r**.

bn_sqr_comba8(**r**, **a**, **b**) operates on the 8 word arrays **a** and **b** and the 16 word array **r**.

The following functions are implemented in C:

bn_cmp_words(**a**, **b**, **n**) operates on the **n** word arrays **a** and **b**. It returns 1, 0 and −1 if **a** is greater than, equal and less than **b**.

bn_mul_normal(**r**, **a**, **na**, **b**, **nb**) operates on the **na** word array **a**, the **nb** word array **b** and the **na**+**nb** word array **r**. It computes **a***b and places the result in **r**.

bn_mul_low_normal(**r**, **a**, **b**, **n**) operates on the **n** word arrays **r**, **a** and **b**. It computes the **n** low words of **a***b and places the result in **r**.

bn_mul_recursive(**r**, **a**, **b**, **n2**, **dna**, **dnb**, **t**) operates on the word arrays **a** and **b** of length **n2**+**dna** and **n2**+**dnb** (**dna** and **dnb** are currently allowed to be 0 or negative) and the 2***n2** word arrays **r** and **t**. **n2** must be a power of 2. It computes **a***b and places the result in **r**.

bn_mul_part_recursive(**r**, **a**, **b**, **n**, **tna**, **tnb**, **tmp**) operates on the word arrays **a** and **b** of length **n**+**tna** and **n**+**tnb** and the 4***n** word arrays **r** and **tmp**.

bn_mul_low_recursive(**r**, **a**, **b**, **n2**, **tmp**) operates on the **n2** word arrays **r** and **tmp** and the **n2**/2 word arrays **a** and **b**.

bn_mul_high(**r**, **a**, **b**, **l**, **n2**, **tmp**) operates on the **n2** word arrays **r**, **a**, **b** and **l** (?) and the 3***n2** word array **tmp**.

BN_mul() calls *bn_mul_normal()*, or an optimized implementation if the factors have the same size: *bn_mul_comba8()* is used if they are 8 words long, *bn_mul_recursive()* if they are larger than **BN_MULL_SIZE_NORMAL** and the size is an exact multiple of the word size, and *bn_mul_part_recursive()* for others that are larger than **BN_MULL_SIZE_NORMAL**.

bn_sqr_normal(**r**, **a**, **n**, **tmp**) operates on the **n** word array **a** and the 2***n** word arrays **tmp** and **r**.

The implementations use the following macros which, depending on the architecture, may use ‘‘long long’’ C operations or inline assembler. They are defined in `bn_lcl.h`.

mul(**r**, **a**, **w**, **c**) computes **w***a+**c** and places the low word of the result in **r** and the high word in **c**.

mul_add(**r**, **a**, **w**, **c**) computes **w***a+**r**+**c** and places the low word of the result in **r** and the high word in **c**.

sqr(**r0**, **r1**, **a**) computes **a***a and places the low word of the result in **r0** and the high word in **r1**.

Size changes

bn_expand() ensures that **b** has enough space for a **bits** bit number. *bn_wexpand()* ensures that **b** has enough space for an **n** word number. If the number has to be expanded, both macros call *bn_expand2()*, which allocates a new **d** array and copies the data. They return **NULL** on error, **b** otherwise.

The *bn_fix_top()* macro reduces **a->top** to point to the most significant non-zero word plus one when **a** has shrunk.

Debugging

bn_check_top() verifies that `((a)->top >= 0 && (a)->top <= (a)->max)`. A violation will cause the program to abort.

bn_print() prints **a** to stderr. *bn_dump()* prints **n** words at **d** (in reverse order, i.e. most significant word first) to stderr.

bn_set_max() makes **a** a static number with a **max** of its current size. This is used by *bn_set_low()* and *bn_set_high()* to make **r** a read-only **BIGNUM** that contains the **n** low or high words of **a**.

If **BN_DEBUG** is not defined, *bn_check_top()*, *bn_print()*, *bn_dump()* and *bn_set_max()* are defined as empty macros.

SEE ALSO

openssl_bn (3)

NAME

BUF_MEM_new, BUF_MEM_free, BUF_MEM_grow, BUF_strdup − simple character arrays structure

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/buffer.h>

BUF_MEM *BUF_MEM_new(void);

void   BUF_MEM_free(BUF_MEM *a);

int    BUF_MEM_grow(BUF_MEM *str, int len);

char * BUF_strdup(const char *str);
```

DESCRIPTION

The buffer library handles simple character arrays. Buffers are used for various purposes in the library, most notably memory BIOs.

The library uses the BUF_MEM structure defined in buffer.h:

```
typedef struct buf_mem_st
{
       int length;     /* current number of bytes */
       char *data;
       int max;        /* size of buffer */
} BUF_MEM;
```

length is the current size of the buffer in bytes, **max** is the amount of memory allocated to the buffer. There are three functions which handle these and one ‘‘miscellaneous’’ function.

BUF_MEM_new() allocates a new buffer of zero size.

BUF_MEM_free() frees up an already existing buffer. The data is zeroed before freeing up in case the buffer contains sensitive data.

BUF_MEM_grow() changes the size of an already existing buffer to **len**. Any data already in the buffer is preserved if it increases in size.

BUF_strdup() copies a null terminated string into a block of allocated memory and returns a pointer to the allocated block. Unlike the standard C library *strdup()* this function uses *OPENSSL_malloc()* and so should be used in preference to the standard library *strdup()* because it can be used for memory leak checking or replacing the *malloc()* function.

The memory allocated from *BUF_strdup()* should be freed up using the *OPENSSL_free()* function.

RETURN VALUES

BUF_MEM_new() returns the buffer or NULL on error.

BUF_MEM_free() has no return value.

BUF_MEM_grow() returns zero on error or the new size (i.e. **len**).

SEE ALSO

openssl_bio (3)

HISTORY

BUF_MEM_new(), BUF_MEM_free() and *BUF_MEM_grow()* are available in all versions of SSLeay and OpenSSL. *BUF_strdup()* was added in SSLeay 0.8.

NAME

DES_random_key, DES_set_key, DES_key_sched, DES_set_key_checked, DES_set_key_unchecked, DES_set_odd_parity, DES_is_weak_key, DES_ecb_encrypt, DES_ecb2_encrypt, DES_ecb3_encrypt, DES_ncbc_encrypt, DES_cfb_encrypt, DES_ofb_encrypt, DES_pcbc_encrypt, DES_cfb64_encrypt, DES_ofb64_encrypt, DES_xcbc_encrypt, DES_ede2_cbc_encrypt, DES_ede2_cfb64_encrypt, DES_ede2_ofb64_encrypt, DES_ede3_cbc_encrypt, DES_ede3_cbcm_encrypt, DES_ede3_cfb64_encrypt, DES_ede3_ofb64_encrypt, DES_cbc_cksum, DES_quad_cksum, DES_string_to_key, DES_string_to_2keys, DES_fcrypt, DES_crypt, DES_enc_read, DES_enc_write − DES encryption

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/des.h>

void DES_random_key(DES_cblock *ret);

int DES_set_key(const_DES_cblock *key, DES_key_schedule *schedule);
int DES_key_sched(const_DES_cblock *key, DES_key_schedule *schedule);
int DES_set_key_checked(const_DES_cblock *key,
       DES_key_schedule *schedule);
void DES_set_key_unchecked(const_DES_cblock *key,
       DES_key_schedule *schedule);

void DES_set_odd_parity(DES_cblock *key);
int DES_is_weak_key(const_DES_cblock *key);

void DES_ecb_encrypt(const_DES_cblock *input, DES_cblock *output,
       DES_key_schedule *ks, int enc);
void DES_ecb2_encrypt(const_DES_cblock *input, DES_cblock *output,
       DES_key_schedule *ks1, DES_key_schedule *ks2, int enc);
void DES_ecb3_encrypt(const_DES_cblock *input, DES_cblock *output,
       DES_key_schedule *ks1, DES_key_schedule *ks2,
       DES_key_schedule *ks3, int enc);

void DES_ncbc_encrypt(const unsigned char *input, unsigned char *output,
       long length, DES_key_schedule *schedule, DES_cblock *ivec,
       int enc);
void DES_cfb_encrypt(const unsigned char *in, unsigned char *out,
       int numbits, long length, DES_key_schedule *schedule,
       DES_cblock *ivec, int enc);
void DES_ofb_encrypt(const unsigned char *in, unsigned char *out,
       int numbits, long length, DES_key_schedule *schedule,
       DES_cblock *ivec);
void DES_pcbc_encrypt(const unsigned char *input, unsigned char *output,
       long length, DES_key_schedule *schedule, DES_cblock *ivec,
       int enc);
void DES_cfb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, DES_key_schedule *schedule, DES_cblock *ivec,
       int *num, int enc);
void DES_ofb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, DES_key_schedule *schedule, DES_cblock *ivec,
       int *num);

void DES_xcbc_encrypt(const unsigned char *input, unsigned char *output,
       long length, DES_key_schedule *schedule, DES_cblock *ivec,
       const_DES_cblock *inw, const_DES_cblock *outw, int enc);
```

```
void DES_ede2_cbc_encrypt(const unsigned char *input,
       unsigned char *output, long length, DES_key_schedule *ks1,
       DES_key_schedule *ks2, DES_cblock *ivec, int enc);
void DES_ede2_cfb64_encrypt(const unsigned char *in,
       unsigned char *out, long length, DES_key_schedule *ks1,
       DES_key_schedule *ks2, DES_cblock *ivec, int *num, int enc);
void DES_ede2_ofb64_encrypt(const unsigned char *in,
       unsigned char *out, long length, DES_key_schedule *ks1,
       DES_key_schedule *ks2, DES_cblock *ivec, int *num);

void DES_ede3_cbc_encrypt(const unsigned char *input,
       unsigned char *output, long length, DES_key_schedule *ks1,
       DES_key_schedule *ks2, DES_key_schedule *ks3, DES_cblock *ivec,
       int enc);
void DES_ede3_cbcm_encrypt(const unsigned char *in, unsigned char *out,
       long length, DES_key_schedule *ks1, DES_key_schedule *ks2,
       DES_key_schedule *ks3, DES_cblock *ivec1, DES_cblock *ivec2,
       int enc);
void DES_ede3_cfb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, DES_key_schedule *ks1, DES_key_schedule *ks2,
       DES_key_schedule *ks3, DES_cblock *ivec, int *num, int enc);
void DES_ede3_ofb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, DES_key_schedule *ks1,
       DES_key_schedule *ks2, DES_key_schedule *ks3,
       DES_cblock *ivec, int *num);

DES_LONG DES_cbc_cksum(const unsigned char *input, DES_cblock *output,
       long length, DES_key_schedule *schedule,
       const_DES_cblock *ivec);
DES_LONG DES_quad_cksum(const unsigned char *input, DES_cblock output[],
       long length, int out_count, DES_cblock *seed);
void DES_string_to_key(const char *str, DES_cblock *key);
void DES_string_to_2keys(const char *str, DES_cblock *key1,
       DES_cblock *key2);

char *DES_fcrypt(const char *buf, const char *salt, char *ret);
char *DES_crypt(const char *buf, const char *salt);

int DES_enc_read(int fd, void *buf, int len, DES_key_schedule *sched,
       DES_cblock *iv);
int DES_enc_write(int fd, const void *buf, int len,
       DES_key_schedule *sched, DES_cblock *iv);
```

DESCRIPTION

This library contains a fast implementation of the DES encryption algorithm.

There are two phases to the use of DES encryption. The first is the generation of a *DES_key_schedule* from a key, the second is the actual encryption. A DES key is of type *DES_cblock*. This type is consists of 8 bytes with odd parity. The least significant bit in each byte is the parity bit. The key schedule is an expanded form of the key; it is used to speed the encryption process.

DES_random_key() generates a random key. The PRNG must be seeded prior to using this function (see *openssl_rand* (3)). If the PRNG could not generate a secure key, 0 is returned.

Before a DES key can be used, it must be converted into the architecture dependent *DES_key_schedule* via the *DES_set_key_checked()* or *DES_set_key_unchecked()* function.

DES_set_key_checked() will check that the key passed is of odd parity and is not a week or semi-weak key. If the parity is wrong, then −1 is returned. If the key is a weak key, then −2 is returned. If an error is returned, the key schedule is not generated.

DES_set_key() works like *DES_set_key_checked()* if the *DES_check_key* flag is non−zero, otherwise like *DES_set_key_unchecked()*. These functions are available for compatibility; it is recommended to use a function that does not depend on a global variable.

DES_set_odd_parity() sets the parity of the passed *key* to odd.

DES_is_weak_key() returns 1 is the passed key is a weak key, 0 if it is ok. The probability that a randomly generated key is weak is 1/2^52, so it is not really worth checking for them.

The following routines mostly operate on an input and output stream of *DES_cblock*s.

DES_ecb_encrypt() is the basic DES encryption routine that encrypts or decrypts a single 8−byte *DES_cblock* in *electronic code book* (ECB) mode. It always transforms the input data, pointed to by *input*, into the output data, pointed to by the *output* argument. If the *encrypt* argument is non-zero (DES_ENCRYPT), the *input* (cleartext) is encrypted in to the *output* (ciphertext) using the key_schedule specified by the *schedule* argument, previously set via *DES_set_key*. If *encrypt* is zero (DES_DECRYPT), the *input* (now ciphertext) is decrypted into the *output* (now cleartext). Input and output may overlap. *DES_ecb_encrypt()* does not return a value.

DES_ecb3_encrypt() encrypts/decrypts the *input* block by using three-key Triple-DES encryption in ECB mode. This involves encrypting the input with *ks1*, decrypting with the key schedule *ks2*, and then encrypting with *ks3*. This routine greatly reduces the chances of brute force breaking of DES and has the advantage of if *ks1*, *ks2* and *ks3* are the same, it is equivalent to just encryption using ECB mode and *ks1* as the key.

The macro *DES_ecb2_encrypt()* is provided to perform two-key Triple-DES encryption by using *ks1* for the final encryption.

DES_ncbc_encrypt() encrypts/decrypts using the *cipher-block-chaining* (CBC) mode of DES. If the *encrypt* argument is non−zero, the routine cipher-block-chain encrypts the cleartext data pointed to by the *input* argument into the ciphertext pointed to by the *output* argument, using the key schedule provided by the *schedule* argument, and initialization vector provided by the *ivec* argument. If the *length* argument is not an integral multiple of eight bytes, the last block is copied to a temporary area and zero filled. The output is always an integral multiple of eight bytes.

DES_xcbc_encrypt() is RSA's DESX mode of DES. It uses *inw* and *outw* to 'whiten' the encryption. *inw* and *outw* are secret (unlike the iv) and are as such, part of the key. So the key is sort of 24 bytes. This is much better than CBC DES.

DES_ede3_cbc_encrypt() implements outer triple CBC DES encryption with three keys. This means that each DES operation inside the CBC mode is really an `C=E(ks3,D(ks2,E(ks1,M)))`. This mode is used by SSL.

The *DES_ede2_cbc_encrypt()* macro implements two-key Triple-DES by reusing *ks1* for the final encryption. `C=E(ks1,D(ks2,E(ks1,M)))`. This form of Triple-DES is used by the RSAREF library.

DES_pcbc_encrypt() encrypt/decrypts using the propagating cipher block chaining mode used by Kerberos v4. Its parameters are the same as *DES_ncbc_encrypt()*.

DES_cfb_encrypt() encrypt/decrypts using cipher feedback mode. This method takes an array of characters as input and outputs and array of characters. It does not require any padding to 8 character groups. Note: the *ivec* variable is changed and the new changed value needs to be passed to the next call to this function. Since this function runs a complete DES ECB encryption per *numbits*, this function is only suggested for use when sending small numbers of characters.

DES_cfb64_encrypt() implements CFB mode of DES with 64bit feedback. Why is this useful you ask? Because this routine will allow you to encrypt an arbitrary number of bytes, no 8 byte padding. Each call to this routine will encrypt the input bytes to output and then update ivec and num. num contains 'how far' we are though ivec. If this does not make much sense, read more about cfb mode of DES :−).

DES_ede3_cfb64_encrypt() and *DES_ede2_cfb64_encrypt()* is the same as *DES_cfb64_encrypt()* except that Triple-DES is used.

DES_ofb_encrypt() encrypts using output feedback mode. This method takes an array of characters as input and outputs and array of characters. It does not require any padding to 8 character groups. Note: the *ivec* variable is changed and the new changed value needs to be passed to the next call to this function. Since this function runs a complete DES ECB encryption per numbits, this function is only suggested for use when sending small numbers of characters.

DES_ofb64_encrypt() is the same as *DES_cfb64_encrypt()* using Output Feed Back mode.

DES_ede3_ofb64_encrypt() and *DES_ede2_ofb64_encrypt()* is the same as *DES_ofb64_encrypt()*, using Triple−DES.

The following functions are included in the DES library for compatibility with the MIT Kerberos library.

DES_cbc_cksum() produces an 8 byte checksum based on the input stream (via CBC encryption). The last 4 bytes of the checksum are returned and the complete 8 bytes are placed in *output*. This function is used by Kerberos v4. Other applications should use *EVP_DigestInit* (3) etc. instead.

DES_quad_cksum() is a Kerberos v4 function. It returns a 4 byte checksum from the input bytes. The algorithm can be iterated over the input, depending on *out_count*, 1, 2, 3 or 4 times. If *output* is non−NULL, the 8 bytes generated by each pass are written into *output*.

The following are DES-based transformations:

DES_fcrypt() is a fast version of the Unix *crypt* (3) function. This version takes only a small amount of space relative to other fast *crypt()* implementations. This is different to the normal crypt in that the third parameter is the buffer that the return value is written into. It needs to be at least 14 bytes long. This function is thread safe, unlike the normal crypt.

DES_crypt() is a faster replacement for the normal system *crypt()*. This function calls *DES_fcrypt()* with a static array passed as the third parameter. This emulates the normal non-thread safe semantics of *crypt* (3).

DES_enc_write() writes *len* bytes to file descriptor *fd* from buffer *buf*. The data is encrypted via *pcbc_encrypt* (default) using *sched* for the key and *iv* as a starting vector. The actual data send down *fd* consists of 4 bytes (in network byte order) containing the length of the following encrypted data. The encrypted data then follows, padded with random data out to a multiple of 8 bytes.

DES_enc_read() is used to read *len* bytes from file descriptor *fd* into buffer *buf*. The data being read from *fd* is assumed to have come from *DES_enc_write()* and is decrypted using *sched* for the key schedule and *iv* for the initial vector.

Warning: The data format used by *DES_enc_write()* and *DES_enc_read()* has a cryptographic weakness: When asked to write more than MAXWRITE bytes, *DES_enc_write()* will split the data into several chunks that are all encrypted using the same IV. So don't use these functions unless you are sure you know what you do (in which case you might not want to use them anyway). They cannot handle non-blocking sockets. *DES_enc_read()* uses an internal state and thus cannot be used on multiple files.

DES_rw_mode is used to specify the encryption mode to use with *DES_enc_read()* and *DES_end_write()*. If set to *DES_PCBC_MODE* (the default), DES_pcbc_encrypt is used. If set to *DES_CBC_MODE* DES_cbc_encrypt is used.

NOTES

Single-key DES is insecure due to its short key size. ECB mode is not suitable for most applications; see *des_modes* (7).

The *openssl_evp* (3) library provides higher-level encryption functions.

BUGS

DES_3cbc_encrypt() is flawed and must not be used in applications.

DES_cbc_encrypt() does not modify **ivec**; use *DES_ncbc_encrypt()* instead.

DES_cfb_encrypt() and *DES_ofb_encrypt()* operates on input of 8 bits. What this means is that if you set numbits to 12, and length to 2, the first 12 bits will come from the 1st input byte and the low half of the second input byte. The second 12 bits will have the low 8 bits taken from the 3rd input byte and the top 4 bits taken from the 4th input byte. The same holds for output. This function has been implemented this way

because most people will be using a multiple of 8 and because once you get into pulling bytes input bytes apart things get ugly!

DES_string_to_key() is available for backward compatibility with the MIT library. New applications should use a cryptographic hash function. The same applies for *DES_string_to_2key()*.

CONFORMING TO

ANSI X3.106

The **des** library was written to be source code compatible with the MIT Kerberos library.

SEE ALSO

crypt (3), *des_modes* (7), *openssl_evp* (3), *openssl_rand* (3)

HISTORY

In OpenSSL 0.9.7, all des_ functions were renamed to DES_ to avoid clashes with older versions of libdes. Compatibility des_ functions are provided for a short while, as well as *crypt()*. Declarations for these are in <openssl/des_old.h>. There is no DES_ variant for *des_random_seed()*. This will happen to other functions as well if they are deemed redundant (*des_random_seed()* just calls *RAND_seed()* and is present for backward compatibility only), buggy or already scheduled for removal.

des_cbc_cksum(), *des_cbc_encrypt()*, *des_ecb_encrypt()*, *des_is_weak_key()*, *des_key_sched()*, *des_pcbc_encrypt()*, *des_quad_cksum()*, *des_random_key()* and *des_string_to_key()* are available in the MIT Kerberos library; *des_check_key_parity()*, *des_fixup_key_parity()* and *des_is_weak_key()* are available in newer versions of that library.

des_set_key_checked() and *des_set_key_unchecked()* were added in OpenSSL 0.9.5.

des_generate_random_block(), *des_init_random_number_generator()*, *des_new_random_key()*, *des_set_random_generator_seed()* and *des_set_sequence_number()* and *des_rand_data()* are used in newer versions of Kerberos but are not implemented here.

des_random_key() generated cryptographically weak random data in SSLeay and in OpenSSL prior version 0.9.5, as well as in the original MIT library.

AUTHOR

Eric Young (eay@cryptsoft.com). Modified for the OpenSSL project (http://www.openssl.org).

NAME

dh − Diffie−Hellman key agreement

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dh.h>
#include <openssl/engine.h>

DH *   DH_new(void);
void   DH_free(DH *dh);

int    DH_size(const DH *dh);

DH *   DH_generate_parameters(int prime_len, int generator,
               void (*callback)(int, int, void *), void *cb_arg);
int    DH_check(const DH *dh, int *codes);

int    DH_generate_key(DH *dh);
int    DH_compute_key(unsigned char *key, BIGNUM *pub_key, DH *dh);

void DH_set_default_method(const DH_METHOD *meth);
const DH_METHOD *DH_get_default_method(void);
int DH_set_method(DH *dh, const DH_METHOD *meth);
DH *DH_new_method(ENGINE *engine);
const DH_METHOD *DH_OpenSSL(void);

int DH_get_ex_new_index(long argl, char *argp, int (*new_func)(),
            int (*dup_func)(), void (*free_func)());
int DH_set_ex_data(DH *d, int idx, char *arg);
char *DH_get_ex_data(DH *d, int idx);

DH *   d2i_DHparams(DH **a, unsigned char **pp, long length);
int    i2d_DHparams(const DH *a, unsigned char **pp);

int    DHparams_print_fp(FILE *fp, const DH *x);
int    DHparams_print(BIO *bp, const DH *x);
```

DESCRIPTION

These functions implement the Diffie-Hellman key agreement protocol. The generation of shared DH parameters is described in *DH_generate_parameters* (3); *DH_generate_key* (3) describes how to perform a key agreement.

The **DH** structure consists of several BIGNUM components.

```
struct
       {
       BIGNUM *p;              // prime number (shared)
       BIGNUM *g;              // generator of Z_p (shared)
       BIGNUM *priv_key;       // private DH value x
       BIGNUM *pub_key;        // public DH value gˆx
       // ...
       };
DH
```

Note that DH keys may use non-standard **DH_METHOD** implementations, either directly or by the use of **ENGINE** modules. In some cases (eg. an ENGINE providing support for hardware-embedded keys), these BIGNUM values will not be used by the implementation or may be used for alternative data storage. For this reason, applications should generally avoid using DH structure elements directly and instead use API functions to query or modify keys.

SEE ALSO

openssl_dhparam (1), *openssl_bn* (3), *openssl_dsa* (3), *openssl_err* (3), *openssl_rand* (3), *openssl_rsa* (3), *engine* (3), *DH_set_method* (3), *DH_new* (3), *DH_get_ex_new_index* (3), *DH_generate_parameters* (3), *DH_compute_key* (3), *d2i_DHparams* (3), *RSA_print* (3)

NAME

dsa − Digital Signature Algorithm

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/dsa.h>
#include <openssl/engine.h>

DSA *  DSA_new(void);
void   DSA_free(DSA *dsa);

int    DSA_size(const DSA *dsa);

DSA *  DSA_generate_parameters(int bits, unsigned char *seed,
               int seed_len, int *counter_ret, unsigned long *h_ret,
               void (*callback)(int, int, void *), void *cb_arg);

DH *   DSA_dup_DH(const DSA *r);

int    DSA_generate_key(DSA *dsa);

int    DSA_sign(int dummy, const unsigned char *dgst, int len,
               unsigned char *sigret, unsigned int *siglen, DSA *dsa);
int    DSA_sign_setup(DSA *dsa, BN_CTX *ctx, BIGNUM **kinvp,
               BIGNUM **rp);
int    DSA_verify(int dummy, const unsigned char *dgst, int len,
               const unsigned char *sigbuf, int siglen, DSA *dsa);

void DSA_set_default_method(const DSA_METHOD *meth);
const DSA_METHOD *DSA_get_default_method(void);
int DSA_set_method(DSA *dsa, const DSA_METHOD *meth);
DSA *DSA_new_method(ENGINE *engine);
const DSA_METHOD *DSA_OpenSSL(void);

int DSA_get_ex_new_index(long argl, char *argp, int (*new_func)(),
           int (*dup_func)(), void (*free_func)());
int DSA_set_ex_data(DSA *d, int idx, char *arg);
char *DSA_get_ex_data(DSA *d, int idx);

DSA_SIG *DSA_SIG_new(void);
void   DSA_SIG_free(DSA_SIG *a);
int    i2d_DSA_SIG(const DSA_SIG *a, unsigned char **pp);
DSA_SIG *d2i_DSA_SIG(DSA_SIG **v, unsigned char **pp, long length);

DSA_SIG *DSA_do_sign(const unsigned char *dgst, int dlen, DSA *dsa);
int    DSA_do_verify(const unsigned char *dgst, int dgst_len,
           DSA_SIG *sig, DSA *dsa);

DSA *  d2i_DSAPublicKey(DSA **a, unsigned char **pp, long length);
DSA *  d2i_DSAPrivateKey(DSA **a, unsigned char **pp, long length);
DSA *  d2i_DSAparams(DSA **a, unsigned char **pp, long length);
int    i2d_DSAPublicKey(const DSA *a, unsigned char **pp);
int    i2d_DSAPrivateKey(const DSA *a, unsigned char **pp);
int    i2d_DSAparams(const DSA *a,unsigned char **pp);

int    DSAparams_print(BIO *bp, const DSA *x);
int    DSAparams_print_fp(FILE *fp, const DSA *x);
int    DSA_print(BIO *bp, const DSA *x, int off);
int    DSA_print_fp(FILE *bp, const DSA *x, int off);
```

DESCRIPTION

These functions implement the Digital Signature Algorithm (DSA). The generation of shared DSA parameters is described in *DSA_generate_parameters* (3); *DSA_generate_key* (3) describes how to generate a signature key. Signature generation and verification are described in *DSA_sign* (3).

The **DSA** structure consists of several BIGNUM components.

```
struct
        {
        BIGNUM *p;              // prime number (public)
        BIGNUM *q;              // 160-bit subprime, q | p-1 (public)
        BIGNUM *g;              // generator of subgroup (public)
        BIGNUM *priv_key;       // private key x
        BIGNUM *pub_key;        // public key y = gˆx
        // ...
        }
DSA;
```

In public keys, **priv_key** is NULL.

Note that DSA keys may use non-standard **DSA_METHOD** implementations, either directly or by the use of **ENGINE** modules. In some cases (eg. an ENGINE providing support for hardware-embedded keys), these BIGNUM values will not be used by the implementation or may be used for alternative data storage. For this reason, applications should generally avoid using DSA structure elements directly and instead use API functions to query or modify keys.

CONFORMING TO

US Federal Information Processing Standard FIPS 186 (Digital Signature Standard, DSS), ANSI X9.30

SEE ALSO

openssl_bn (3), *openssl_dh* (3), *openssl_err* (3), *openssl_rand* (3), *openssl_rsa* (3), *openssl_sha* (3), *engine* (3), *DSA_new* (3), *DSA_size* (3), *DSA_generate_parameters* (3), *DSA_dup_DH* (3), *DSA_generate_key* (3), *DSA_sign* (3), *DSA_set_method* (3), *DSA_get_ex_new_index* (3), *RSA_print* (3)

NAME

ecdsa − Elliptic Curve Digital Signature Algorithm

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ecdsa.h>

ECDSA_SIG*      ECDSA_SIG_new(void);
void            ECDSA_SIG_free(ECDSA_SIG *sig);
int             i2d_ECDSA_SIG(const ECDSA_SIG *sig, unsigned char **pp);
ECDSA_SIG*      d2i_ECDSA_SIG(ECDSA_SIG **sig, const unsigned char **pp,
                long len);

ECDSA_SIG*      ECDSA_do_sign(const unsigned char *dgst, int dgst_len,
                        EC_KEY *eckey);
ECDSA_SIG*      ECDSA_do_sign_ex(const unsigned char *dgst, int dgstlen,
                        const BIGNUM *kinv, const BIGNUM *rp,
                        EC_KEY *eckey);
int             ECDSA_do_verify(const unsigned char *dgst, int dgst_len,
                        const ECDSA_SIG *sig, EC_KEY* eckey);
int             ECDSA_sign_setup(EC_KEY *eckey, BN_CTX *ctx,
                        BIGNUM **kinv, BIGNUM **rp);
int             ECDSA_sign(int type, const unsigned char *dgst,
                        int dgstlen, unsigned char *sig,
                        unsigned int *siglen, EC_KEY *eckey);
int             ECDSA_sign_ex(int type, const unsigned char *dgst,
                        int dgstlen, unsigned char *sig,
                        unsigned int *siglen, const BIGNUM *kinv,
                        const BIGNUM *rp, EC_KEY *eckey);
int             ECDSA_verify(int type, const unsigned char *dgst,
                        int dgstlen, const unsigned char *sig,
                        int siglen, EC_KEY *eckey);
int             ECDSA_size(const EC_KEY *eckey);

const ECDSA_METHOD*     ECDSA_OpenSSL(void);
void            ECDSA_set_default_method(const ECDSA_METHOD *meth);
const ECDSA_METHOD*     ECDSA_get_default_method(void);
int             ECDSA_set_method(EC_KEY *eckey,const ECDSA_METHOD *meth);

int             ECDSA_get_ex_new_index(long argl, void *argp,
                        CRYPTO_EX_new *new_func,
                        CRYPTO_EX_dup *dup_func,
                        CRYPTO_EX_free *free_func);
int             ECDSA_set_ex_data(EC_KEY *d, int idx, void *arg);
void*           ECDSA_get_ex_data(EC_KEY *d, int idx);
```

DESCRIPTION

The **ECDSA_SIG** structure consists of two BIGNUMs for the r and s value of a ECDSA signature (see X9.62 or FIPS 186−2).

```
struct
        {
        BIGNUM *r;
        BIGNUM *s;
} ECDSA_SIG;
```

ECDSA_SIG_new() allocates a new **ECDSA_SIG** structure (note: this function also allocates the BIGNUMs) and initialize it.

ECDSA_SIG_free() frees the **ECDSA_SIG** structure **sig**.

i2d_ECDSA_SIG() creates the DER encoding of the ECDSA signature **sig** and writes the encoded signature to ***pp** (note: if **pp** is NULL **i2d_ECDSA_SIG** returns the expected length in bytes of the DER encoded signature). **i2d_ECDSA_SIG** returns the length of the DER encoded signature (or 0 on error).

d2i_ECDSA_SIG() decodes a DER encoded ECDSA signature and returns the decoded signature in a newly allocated **ECDSA_SIG** structure. ***sig** points to the buffer containing the DER encoded signature of size **len**.

ECDSA_size() returns the maximum length of a DER encoded ECDSA signature created with the private EC key **eckey**.

ECDSA_sign_setup() may be used to precompute parts of the signing operation. **eckey** is the private EC key and **ctx** is a pointer to **BN_CTX** structure (or NULL). The precomputed values or returned in **kinv** and **rp** and can be used in a later call to **ECDSA_sign_ex** or **ECDSA_do_sign_ex**.

ECDSA_sign() is wrapper function for ECDSA_sign_ex with **kinv** and **rp** set to NULL.

ECDSA_sign_ex() computes a digital signature of the **dgstlen** bytes hash value **dgst** using the private EC key **eckey** and the optional pre-computed values **kinv** and **rp**. The DER encoded signatures is stored in **sig** and it's length is returned in **sig_len**. Note: **sig** must point to **ECDSA_size** bytes of memory. The parameter **type** is ignored.

ECDSA_verify() verifies that the signature in **sig** of size **siglen** is a valid ECDSA signature of the hash value value **dgst** of size **dgstlen** using the public key **eckey**. The parameter **type** is ignored.

ECDSA_do_sign() is wrapper function for ECDSA_do_sign_ex with **kinv** and **rp** set to NULL.

ECDSA_do_sign_ex() computes a digital signature of the **dgst_len** bytes hash value **dgst** using the private key **eckey** and the optional pre-computed values **kinv** and **rp**. The signature is returned in a newly allocated **ECDSA_SIG** structure (or NULL on error).

ECDSA_do_verify() verifies that the signature **sig** is a valid ECDSA signature of the hash value **dgst** of size **dgst_len** using the public key **eckey**.

RETURN VALUES

ECDSA_size() returns the maximum length signature or 0 on error.

ECDSA_sign_setup() and *ECDSA_sign()* return 1 if successful or −1 on error.

ECDSA_verify() and *ECDSA_do_verify()* return 1 for a valid signature, 0 for an invalid signature and −1 on error. The error codes can be obtained by *ERR_get_error* (3).

EXAMPLES

Creating a ECDSA signature of given SHA−1 hash value using the named curve secp192k1.

First step: create a EC_KEY object (note: this part is **not** ECDSA specific)

```
int        ret;
ECDSA_SIG *sig;
EC_KEY    *eckey = EC_KEY_new();
if (eckey == NULL)
       {
       /* error */
       }
key->group = EC_GROUP_new_by_nid(NID_secp192k1);
if (key->group == NULL)
       {
       /* error */
       }
if (!EC_KEY_generate_key(eckey))
       {
       /* error */
       }
```

Second step: compute the ECDSA signature of a SHA−1 hash value using **ECDSA_do_sign**

```
sig = ECDSA_do_sign(digest, 20, eckey);
if (sig == NULL)
       {
       /* error */
       }
```

or using **ECDSA_sign**

```
unsigned char *buffer, *pp;
int            buf_len;
buf_len = ECDSA_size(eckey);
buffer  = OPENSSL_malloc(buf_len);
pp = buffer;
if (!ECDSA_sign(0, dgst, dgstlen, pp, &buf_len, eckey);
       {
       /* error */
       }
```

Third step: verify the created ECDSA signature using **ECDSA_do_verify**

```
ret = ECDSA_do_verify(digest, 20, sig, eckey);
```

or using **ECDSA_verify**

```
ret = ECDSA_verify(0, digest, 20, buffer, buf_len, eckey);
```

and finally evaluate the return value:

```
if (ret == -1)
       {
       /* error */
       }
else if (ret == 0)
       {
       /* incorrect signature */
       }
else   /* ret == 1 */
       {
       /* signature ok */
       }
```

CONFORMING TO

ANSI X9.62, US Federal Information Processing Standard FIPS 186−2 (Digital Signature Standard, DSS)

SEE ALSO

openssl_dsa (3), *openssl_rsa* (3)

HISTORY

The ecdsa implementation was first introduced in OpenSSL 0.9.8

AUTHOR

Nils Larsch for the OpenSSL project (http://www.openssl.org).

NAME

engine − ENGINE cryptographic module support

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/engine.h>

ENGINE *ENGINE_get_first(void);
ENGINE *ENGINE_get_last(void);
ENGINE *ENGINE_get_next(ENGINE *e);
ENGINE *ENGINE_get_prev(ENGINE *e);

int ENGINE_add(ENGINE *e);
int ENGINE_remove(ENGINE *e);

ENGINE *ENGINE_by_id(const char *id);

int ENGINE_init(ENGINE *e);
int ENGINE_finish(ENGINE *e);

void ENGINE_load_openssl(void);
void ENGINE_load_dynamic(void);
#ifndef OPENSSL_NO_STATIC_ENGINE
void ENGINE_load_4758cca(void);
void ENGINE_load_aep(void);
void ENGINE_load_atalla(void);
void ENGINE_load_chil(void);
void ENGINE_load_cswift(void);
void ENGINE_load_gmp(void);
void ENGINE_load_nuron(void);
void ENGINE_load_sureware(void);
void ENGINE_load_ubsec(void);
#endif
void ENGINE_load_cryptodev(void);
void ENGINE_load_builtin_engines(void);

void ENGINE_cleanup(void);

ENGINE *ENGINE_get_default_RSA(void);
ENGINE *ENGINE_get_default_DSA(void);
ENGINE *ENGINE_get_default_ECDH(void);
ENGINE *ENGINE_get_default_ECDSA(void);
ENGINE *ENGINE_get_default_DH(void);
ENGINE *ENGINE_get_default_RAND(void);
ENGINE *ENGINE_get_cipher_engine(int nid);
ENGINE *ENGINE_get_digest_engine(int nid);

int ENGINE_set_default_RSA(ENGINE *e);
int ENGINE_set_default_DSA(ENGINE *e);
int ENGINE_set_default_ECDH(ENGINE *e);
int ENGINE_set_default_ECDSA(ENGINE *e);
int ENGINE_set_default_DH(ENGINE *e);
int ENGINE_set_default_RAND(ENGINE *e);
int ENGINE_set_default_ciphers(ENGINE *e);
int ENGINE_set_default_digests(ENGINE *e);
int ENGINE_set_default_string(ENGINE *e, const char *list);

int ENGINE_set_default(ENGINE *e, unsigned int flags);
```

```
unsigned int ENGINE_get_table_flags(void);
void ENGINE_set_table_flags(unsigned int flags);

int ENGINE_register_RSA(ENGINE *e);
void ENGINE_unregister_RSA(ENGINE *e);
void ENGINE_register_all_RSA(void);
int ENGINE_register_DSA(ENGINE *e);
void ENGINE_unregister_DSA(ENGINE *e);
void ENGINE_register_all_DSA(void);
int ENGINE_register_ECDH(ENGINE *e);
void ENGINE_unregister_ECDH(ENGINE *e);
void ENGINE_register_all_ECDH(void);
int ENGINE_register_ECDSA(ENGINE *e);
void ENGINE_unregister_ECDSA(ENGINE *e);
void ENGINE_register_all_ECDSA(void);
int ENGINE_register_DH(ENGINE *e);
void ENGINE_unregister_DH(ENGINE *e);
void ENGINE_register_all_DH(void);
int ENGINE_register_RAND(ENGINE *e);
void ENGINE_unregister_RAND(ENGINE *e);
void ENGINE_register_all_RAND(void);
int ENGINE_register_STORE(ENGINE *e);
void ENGINE_unregister_STORE(ENGINE *e);
void ENGINE_register_all_STORE(void);
int ENGINE_register_ciphers(ENGINE *e);
void ENGINE_unregister_ciphers(ENGINE *e);
void ENGINE_register_all_ciphers(void);
int ENGINE_register_digests(ENGINE *e);
void ENGINE_unregister_digests(ENGINE *e);
void ENGINE_register_all_digests(void);
int ENGINE_register_complete(ENGINE *e);
int ENGINE_register_all_complete(void);

int ENGINE_ctrl(ENGINE *e, int cmd, long i, void *p, void (*f)(void));
int ENGINE_cmd_is_executable(ENGINE *e, int cmd);
int ENGINE_ctrl_cmd(ENGINE *e, const char *cmd_name,
        long i, void *p, void (*f)(void), int cmd_optional);
int ENGINE_ctrl_cmd_string(ENGINE *e, const char *cmd_name, const char *arg,
        int cmd_optional);

int ENGINE_set_ex_data(ENGINE *e, int idx, void *arg);
void *ENGINE_get_ex_data(const ENGINE *e, int idx);

int ENGINE_get_ex_new_index(long argl, void *argp, CRYPTO_EX_new *new_func,
        CRYPTO_EX_dup *dup_func, CRYPTO_EX_free *free_func);

ENGINE *ENGINE_new(void);
int ENGINE_free(ENGINE *e);
int ENGINE_up_ref(ENGINE *e);
```

```
int ENGINE_set_id(ENGINE *e, const char *id);
int ENGINE_set_name(ENGINE *e, const char *name);
int ENGINE_set_RSA(ENGINE *e, const RSA_METHOD *rsa_meth);
int ENGINE_set_DSA(ENGINE *e, const DSA_METHOD *dsa_meth);
int ENGINE_set_ECDH(ENGINE *e, const ECDH_METHOD *dh_meth);
int ENGINE_set_ECDSA(ENGINE *e, const ECDSA_METHOD *dh_meth);
int ENGINE_set_DH(ENGINE *e, const DH_METHOD *dh_meth);
int ENGINE_set_RAND(ENGINE *e, const RAND_METHOD *rand_meth);
int ENGINE_set_STORE(ENGINE *e, const STORE_METHOD *rand_meth);
int ENGINE_set_destroy_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR destroy_f);
int ENGINE_set_init_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR init_f);
int ENGINE_set_finish_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR finish_f);
int ENGINE_set_ctrl_function(ENGINE *e, ENGINE_CTRL_FUNC_PTR ctrl_f);
int ENGINE_set_load_privkey_function(ENGINE *e, ENGINE_LOAD_KEY_PTR loadpriv_f);
int ENGINE_set_load_pubkey_function(ENGINE *e, ENGINE_LOAD_KEY_PTR loadpub_f);
int ENGINE_set_ciphers(ENGINE *e, ENGINE_CIPHERS_PTR f);
int ENGINE_set_digests(ENGINE *e, ENGINE_DIGESTS_PTR f);
int ENGINE_set_flags(ENGINE *e, int flags);
int ENGINE_set_cmd_defns(ENGINE *e, const ENGINE_CMD_DEFN *defns);

const char *ENGINE_get_id(const ENGINE *e);
const char *ENGINE_get_name(const ENGINE *e);
const RSA_METHOD *ENGINE_get_RSA(const ENGINE *e);
const DSA_METHOD *ENGINE_get_DSA(const ENGINE *e);
const ECDH_METHOD *ENGINE_get_ECDH(const ENGINE *e);
const ECDSA_METHOD *ENGINE_get_ECDSA(const ENGINE *e);
const DH_METHOD *ENGINE_get_DH(const ENGINE *e);
const RAND_METHOD *ENGINE_get_RAND(const ENGINE *e);
const STORE_METHOD *ENGINE_get_STORE(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_destroy_function(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_init_function(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_finish_function(const ENGINE *e);
ENGINE_CTRL_FUNC_PTR ENGINE_get_ctrl_function(const ENGINE *e);
ENGINE_LOAD_KEY_PTR ENGINE_get_load_privkey_function(const ENGINE *e);
ENGINE_LOAD_KEY_PTR ENGINE_get_load_pubkey_function(const ENGINE *e);
ENGINE_CIPHERS_PTR ENGINE_get_ciphers(const ENGINE *e);
ENGINE_DIGESTS_PTR ENGINE_get_digests(const ENGINE *e);
const EVP_CIPHER *ENGINE_get_cipher(ENGINE *e, int nid);
const EVP_MD *ENGINE_get_digest(ENGINE *e, int nid);
int ENGINE_get_flags(const ENGINE *e);
const ENGINE_CMD_DEFN *ENGINE_get_cmd_defns(const ENGINE *e);

EVP_PKEY *ENGINE_load_private_key(ENGINE *e, const char *key_id,
    UI_METHOD *ui_method, void *callback_data);
EVP_PKEY *ENGINE_load_public_key(ENGINE *e, const char *key_id,
    UI_METHOD *ui_method, void *callback_data);

void ENGINE_add_conf_module(void);
```

DESCRIPTION

These functions create, manipulate, and use cryptographic modules in the form of **ENGINE** objects. These objects act as containers for implementations of cryptographic algorithms, and support a reference-counted mechanism to allow them to be dynamically loaded in and out of the running application.

The cryptographic functionality that can be provided by an **ENGINE** implementation includes the following abstractions;

```
RSA_METHOD - for providing alternative RSA implementations
DSA_METHOD, DH_METHOD, RAND_METHOD, ECDH_METHOD, ECDSA_METHOD,
      STORE_METHOD - similarly for other OpenSSL APIs
EVP_CIPHER - potentially multiple cipher algorithms (indexed by 'nid')
EVP_DIGEST - potentially multiple hash algorithms (indexed by 'nid')
key-loading - loading public and/or private EVP_PKEY keys
```

Reference counting and handles

Due to the modular nature of the ENGINE API, pointers to ENGINEs need to be treated as handles − ie. not only as pointers, but also as references to the underlying ENGINE object. Ie. one should obtain a new reference when making copies of an ENGINE pointer if the copies will be used (and released) independently.

ENGINE objects have two levels of reference-counting to match the way in which the objects are used. At the most basic level, each ENGINE pointer is inherently a **structural** reference − a structural reference is required to use the pointer value at all, as this kind of reference is a guarantee that the structure can not be deallocated until the reference is released.

However, a structural reference provides no guarantee that the ENGINE is initiliased and able to use any of its cryptographic implementations. Indeed it's quite possible that most ENGINEs will not initialise at all in typical environments, as ENGINEs are typically used to support specialised hardware. To use an ENGINE's functionality, you need a **functional** reference. This kind of reference can be considered a specialised form of structural reference, because each functional reference implicitly contains a structural reference as well − however to avoid difficult-to-find programming bugs, it is recommended to treat the two kinds of reference independently. If you have a functional reference to an ENGINE, you have a guarantee that the ENGINE has been initialised ready to perform cryptographic operations and will remain uninitialised until after you have released your reference.

Structural references

This basic type of reference is used for instantiating new ENGINEs, iterating across OpenSSL's internal linked-list of loaded ENGINEs, reading information about an ENGINE, etc. Essentially a structural reference is sufficient if you only need to query or manipulate the data of an ENGINE implementation rather than use its functionality.

The *ENGINE_new()* function returns a structural reference to a new (empty) ENGINE object. There are other ENGINE API functions that return structural references such as; *ENGINE_by_id()*, *ENGINE_get_first()*, *ENGINE_get_last()*, *ENGINE_get_next()*, *ENGINE_get_prev()*. All structural references should be released by a corresponding to call to the *ENGINE_free()* function − the ENGINE object itself will only actually be cleaned up and deallocated when the last structural reference is released.

It should also be noted that many ENGINE API function calls that accept a structural reference will internally obtain another reference − typically this happens whenever the supplied ENGINE will be needed by OpenSSL after the function has returned. Eg. the function to add a new ENGINE to OpenSSL's internal list is *ENGINE_add()* − if this function returns success, then OpenSSL will have stored a new structural reference internally so the caller is still responsible for freeing their own reference with *ENGINE_free()* when they are finished with it. In a similar way, some functions will automatically release the structural reference passed to it if part of the function's job is to do so. Eg. the *ENGINE_get_next()* and *ENGINE_get_prev()* functions are used for iterating across the internal ENGINE list − they will return a new structural reference to the next (or previous) ENGINE in the list or NULL if at the end (or beginning) of the list, but in either case the structural reference passed to the function is released on behalf of the caller.

To clarify a particular function's handling of references, one should always consult that function's documentation ''man'' page, or failing that the openssl/engine.h header file includes some hints.

Functional references

As mentioned, functional references exist when the cryptographic functionality of an ENGINE is required to be available. A functional reference can be obtained in one of two ways; from an existing structural reference to the required ENGINE, or by asking OpenSSL for the default operational ENGINE for a given cryptographic purpose.

To obtain a functional reference from an existing structural reference, call the *ENGINE_init()* function. This returns zero if the ENGINE was not already operational and couldn't be successfully initialised (eg. lack of system drivers, no special hardware attached, etc), otherwise it will return non-zero to indicate that the ENGINE is now operational and will have allocated a new **functional** reference to the ENGINE. All functional references are released by calling *ENGINE_finish()* (which removes the implicit structural reference as well).

The second way to get a functional reference is by asking OpenSSL for a default implementation for a given task, eg. by *ENGINE_get_default_RSA()*, *ENGINE_get_default_cipher_engine()*, etc. These are discussed in the next section, though they are not usually required by application programmers as they are used automatically when creating and using the relevant algorithm-specific types in OpenSSL, such as RSA, DSA, EVP_CIPHER_CTX, etc.

Default implementations

For each supported abstraction, the ENGINE code maintains an internal table of state to control which implementations are available for a given abstraction and which should be used by default. These implementations are registered in the tables and indexed by an 'nid' value, because abstractions like EVP_CIPHER and EVP_DIGEST support many distinct algorithms and modes, and ENGINEs can support arbitrarily many of them. In the case of other abstractions like RSA, DSA, etc, there is only one ''algorithm'' so all implementations implicitly register using the same 'nid' index.

When a default ENGINE is requested for a given abstraction/algorithm/mode, (eg. when calling RSA_new_method(NULL)), a ''get_default'' call will be made to the ENGINE subsystem to process the corresponding state table and return a functional reference to an initialised ENGINE whose implementation should be used. If no ENGINE should (or can) be used, it will return NULL and the caller will operate with a NULL ENGINE handle − this usually equates to using the conventional software implementation. In the latter case, OpenSSL will from then on behave the way it used to before the ENGINE API existed.

Each state table has a flag to note whether it has processed this ''get_default'' query since the table was last modified, because to process this question it must iterate across all the registered ENGINEs in the table trying to initialise each of them in turn, in case one of them is operational. If it returns a functional reference to an ENGINE, it will also cache another reference to speed up processing future queries (without needing to iterate across the table). Likewise, it will cache a NULL response if no ENGINE was available so that future queries won't repeat the same iteration unless the state table changes. This behaviour can also be changed; if the ENGINE_TABLE_FLAG_NOINIT flag is set (using *ENGINE_set_table_flags()*), no attempted initialisations will take place, instead the only way for the state table to return a non-NULL ENGINE to the ''get_default'' query will be if one is expressly set in the table. Eg. *ENGINE_set_default_RSA()* does the same job as *ENGINE_register_RSA()* except that it also sets the state table's cached response for the ''get_default'' query. In the case of abstractions like EVP_CIPHER, where implementations are indexed by 'nid', these flags and cached-responses are distinct for each 'nid' value.

Application requirements

This section will explain the basic things an application programmer should support to make the most useful elements of the ENGINE functionality available to the user. The first thing to consider is whether the programmer wishes to make alternative ENGINE modules available to the application and user. OpenSSL maintains an internal linked list of ''visible'' ENGINEs from which it has to operate − at start−up, this list is empty and in fact if an application does not call any ENGINE API calls and it uses static linking against openssl, then the resulting application binary will not contain any alternative ENGINE code at all. So the first consideration is whether any/all available ENGINE implementations should be made visible to OpenSSL − this is controlled by calling the various ''load'' functions, eg.

```
/* Make the "dynamic" ENGINE available */
void ENGINE_load_dynamic(void);
/* Make the CryptoSwift hardware acceleration support available */
void ENGINE_load_cswift(void);
/* Make support for nCipher's "CHIL" hardware available */
void ENGINE_load_chil(void);
...
/* Make ALL ENGINE implementations bundled with OpenSSL available */
void ENGINE_load_builtin_engines(void);
```

Having called any of these functions, ENGINE objects would have been dynamically allocated and populated with these implementations and linked into OpenSSL's internal linked list. At this point it is important to mention an important API function;

```
void ENGINE_cleanup(void);
```

If no ENGINE API functions are called at all in an application, then there are no inherent memory leaks to worry about from the ENGINE functionality, however if any ENGINEs are loaded, even if they are never registered or used, it is necessary to use the *ENGINE_cleanup()* function to correspondingly cleanup before program exit, if the caller wishes to avoid memory leaks. This mechanism uses an internal callback registration table so that any ENGINE API functionality that knows it requires cleanup can register its cleanup details to be called during *ENGINE_cleanup()*. This approach allows *ENGINE_cleanup()* to clean up after any ENGINE functionality at all that your program uses, yet doesn't automatically create linker dependencies to all possible ENGINE functionality − only the cleanup callbacks required by the functionality you do use will be required by the linker.

The fact that ENGINEs are made visible to OpenSSL (and thus are linked into the program and loaded into memory at run−time) does not mean they are ''registered'' or called into use by OpenSSL automatically − that behaviour is something for the application to control. Some applications will want to allow the user to specify exactly which ENGINE they want used if any is to be used at all. Others may prefer to load all support and have OpenSSL automatically use at run-time any ENGINE that is able to successfully initialise − ie. to assume that this corresponds to acceleration hardware attached to the machine or some such thing. There are probably numerous other ways in which applications may prefer to handle things, so we will simply illustrate the consequences as they apply to a couple of simple cases and leave developers to consider these and the source code to openssl's builtin utilities as guides.

Using a specific ENGINE implementation

Here we'll assume an application has been configured by its user or admin to want to use the ''ACME'' ENGINE if it is available in the version of OpenSSL the application was compiled with. If it is available, it should be used by default for all RSA, DSA, and symmetric cipher operation, otherwise OpenSSL should use its builtin software as per usual. The following code illustrates how to approach this;

```
ENGINE *e;
const char *engine_id = "ACME";
ENGINE_load_builtin_engines();
e = ENGINE_by_id(engine_id);
if(!e)
    /* the engine isn't available */
    return;
if(!ENGINE_init(e)) {
    /* the engine couldn't initialise, release 'e' */
    ENGINE_free(e);
    return;
}
if(!ENGINE_set_default_RSA(e))
    /* This should only happen when 'e' can't initialise, but the previous
     * statement suggests it did. */
    abort();
ENGINE_set_default_DSA(e);
ENGINE_set_default_ciphers(e);
/* Release the functional reference from ENGINE_init() */
ENGINE_finish(e);
/* Release the structural reference from ENGINE_by_id() */
ENGINE_free(e);
```

Automatically using builtin ENGINE implementations

Here we'll assume we want to load and register all ENGINE implementations bundled with OpenSSL, such that for any cryptographic algorithm required by OpenSSL − if there is an ENGINE that implements it and can be initialise, it should be used. The following code illustrates how this can work;

```
/* Load all bundled ENGINEs into memory and make them visible */
ENGINE_load_builtin_engines();
/* Register all of them for every algorithm they collectively implement */
ENGINE_register_all_complete();
```

That's all that's required. Eg. the next time OpenSSL tries to set up an RSA key, any bundled ENGINEs that implement RSA_METHOD will be passed to *ENGINE_init()* and if any of those succeed, that ENGINE will be set as the default for RSA use from then on.

Advanced configuration support

There is a mechanism supported by the ENGINE framework that allows each ENGINE implementation to define an arbitrary set of configuration ''commands'' and expose them to OpenSSL and any applications based on OpenSSL. This mechanism is entirely based on the use of name-value pairs and assumes ASCII input (no unicode or UTF for now!), so it is ideal if applications want to provide a transparent way for users to provide arbitrary configuration ''directives'' directly to such ENGINEs. It is also possible for the application to dynamically interrogate the loaded ENGINE implementations for the names, descriptions, and input flags of their available ''control commands'', providing a more flexible configuration scheme. However, if the user is expected to know which ENGINE device he/she is using (in the case of specialised hardware, this goes without saying) then applications may not need to concern themselves with discovering the supported control commands and simply prefer to pass settings into ENGINEs exactly as they are provided by the user.

Before illustrating how control commands work, it is worth mentioning what they are typically used for. Broadly speaking there are two uses for control commands; the first is to provide the necessary details to the implementation (which may know nothing at all specific to the host system) so that it can be initialised for use. This could include the path to any driver or config files it needs to load, required network addresses, smart-card identifiers, passwords to initialise protected devices, logging information, etc etc. This class of commands typically needs to be passed to an ENGINE **before** attempting to initialise it, ie.

before calling *ENGINE_init()*. The other class of commands consist of settings or operations that tweak certain behaviour or cause certain operations to take place, and these commands may work either before or after *ENGINE_init()*, or in some cases both. ENGINE implementations should provide indications of this in the descriptions attached to builtin control commands and/or in external product documentation.

Issuing control commands to an ENGINE

Let's illustrate by example; a function for which the caller supplies the name of the ENGINE it wishes to use, a table of string-pairs for use before initialisation, and another table for use after initialisation. Note that the string-pairs used for control commands consist of a command ''name'' followed by the command ''parameter'' − the parameter could be NULL in some cases but the name can not. This function should initialise the ENGINE (issuing the ''pre'' commands beforehand and the ''post'' commands afterwards) and set it as the default for everything except RAND and then return a boolean success or failure.

```
int generic_load_engine_fn(const char *engine_id,
                           const char **pre_cmds, int pre_num,
                           const char **post_cmds, int post_num)
{
    ENGINE *e = ENGINE_by_id(engine_id);
    if(!e) return 0;
    while(pre_num--) {
        if(!ENGINE_ctrl_cmd_string(e, pre_cmds[0], pre_cmds[1], 0)) {
            fprintf(stderr, "Failed command (%s - %s:%s)\n", engine_id,
                pre_cmds[0], pre_cmds[1] ? pre_cmds[1] : "(NULL)");
            ENGINE_free(e);
            return 0;
        }
        pre_cmds += 2;
    }
    if(!ENGINE_init(e)) {
        fprintf(stderr, "Failed initialisation\n");
        ENGINE_free(e);
        return 0;
    }
    /* ENGINE_init() returned a functional reference, so free the structural
     * reference from ENGINE_by_id(). */
    ENGINE_free(e);
    while(post_num--) {
        if(!ENGINE_ctrl_cmd_string(e, post_cmds[0], post_cmds[1], 0)) {
            fprintf(stderr, "Failed command (%s - %s:%s)\n", engine_id,
                post_cmds[0], post_cmds[1] ? post_cmds[1] : "(NULL)");
            ENGINE_finish(e);
            return 0;
        }
        post_cmds += 2;
    }
    ENGINE_set_default(e, ENGINE_METHOD_ALL & ~ENGINE_METHOD_RAND);
    /* Success */
    return 1;
}
```

Note that *ENGINE_ctrl_cmd_string()* accepts a boolean argument that can relax the semantics of the function − if set non-zero it will only return failure if the ENGINE supported the given command name but failed while executing it, if the ENGINE doesn't support the command name it will simply return success without doing anything. In this case we assume the user is only supplying commands specific to the given ENGINE so we set this to FALSE.

Discovering supported control commands

It is possible to discover at run-time the names, numerical−ids, descriptions and input parameters of the control commands supported by an ENGINE using a structural reference. Note that some control commands are defined by OpenSSL itself and it will intercept and handle these control commands on behalf of the ENGINE, ie. the ENGINE's *ctrl()* handler is not used for the control command. openssl/engine.h defines an index, ENGINE_CMD_BASE, that all control commands implemented by ENGINEs should be numbered from. Any command value lower than this symbol is considered a ''generic'' command is handled directly by the OpenSSL core routines.

It is using these ''core'' control commands that one can discover the the control commands implemented by a given ENGINE, specifically the commands;

```
#define ENGINE_HAS_CTRL_FUNCTION              10
#define ENGINE_CTRL_GET_FIRST_CMD_TYPE        11
#define ENGINE_CTRL_GET_NEXT_CMD_TYPE         12
#define ENGINE_CTRL_GET_CMD_FROM_NAME         13
#define ENGINE_CTRL_GET_NAME_LEN_FROM_CMD     14
#define ENGINE_CTRL_GET_NAME_FROM_CMD         15
#define ENGINE_CTRL_GET_DESC_LEN_FROM_CMD     16
#define ENGINE_CTRL_GET_DESC_FROM_CMD         17
#define ENGINE_CTRL_GET_CMD_FLAGS             18
```

Whilst these commands are automatically processed by the OpenSSL framework code, they use various properties exposed by each ENGINE to process these queries. An ENGINE has 3 properties it exposes that can affect how this behaves; it can supply a *ctrl()* handler, it can specify ENGINE_FLAGS_MANUAL_CMD_CTRL in the ENGINE's flags, and it can expose an array of control command descriptions. If an ENGINE specifies the ENGINE_FLAGS_MANUAL_CMD_CTRL flag, then it will simply pass all these ''core'' control commands directly to the ENGINE's *ctrl()* handler (and thus, it must have supplied one), so it is up to the ENGINE to reply to these ''discovery'' commands itself. If that flag is not set, then the OpenSSL framework code will work with the following rules;

```
if no ctrl() handler supplied;
    ENGINE_HAS_CTRL_FUNCTION returns FALSE (zero),
    all other commands fail.
if a ctrl() handler was supplied but no array of control commands;
    ENGINE_HAS_CTRL_FUNCTION returns TRUE,
    all other commands fail.
if a ctrl() handler and array of control commands was supplied;
    ENGINE_HAS_CTRL_FUNCTION returns TRUE,
    all other commands proceed processing ...
```

If the ENGINE's array of control commands is empty then all other commands will fail, otherwise; ENGINE_CTRL_GET_FIRST_CMD_TYPE returns the identifier of the first command supported by the ENGINE, ENGINE_GET_NEXT_CMD_TYPE takes the identifier of a command supported by the ENGINE and returns the next command identifier or fails if there are no more, ENGINE_CMD_FROM_NAME takes a string name for a command and returns the corresponding identifier or fails if no such command name exists, and the remaining commands take a command identifier and return properties of the corresponding commands. All except ENGINE_CTRL_GET_FLAGS return the string length of a command name or description, or populate a supplied character buffer with a copy of the command name or description. ENGINE_CTRL_GET_FLAGS returns a bitwise−OR'd mask of the following possible values;

```
#define ENGINE_CMD_FLAG_NUMERIC               (unsigned int)0x0001
#define ENGINE_CMD_FLAG_STRING                (unsigned int)0x0002
#define ENGINE_CMD_FLAG_NO_INPUT              (unsigned int)0x0004
#define ENGINE_CMD_FLAG_INTERNAL              (unsigned int)0x0008
```

If the ENGINE_CMD_FLAG_INTERNAL flag is set, then any other flags are purely informational to the caller − this flag will prevent the command being usable for any higher-level ENGINE functions such as

ENGINE_ctrl_cmd_string(). ''INTERNAL'' commands are not intended to be exposed to text-based configuration by applications, administrations, users, etc. These can support arbitrary operations via *ENGINE_ctrl()*, including passing to and/or from the control commands data of any arbitrary type. These commands are supported in the discovery mechanisms simply to allow applications determinie if an ENGINE supports certain specific commands it might want to use (eg. application ''foo'' might query various ENGINEs to see if they implement ''FOO_GET_VENDOR_LOGO_GIF'' − and ENGINE could therefore decide whether or not to support this ''foo''−specific extension).

Future developments

The ENGINE API and internal architecture is currently being reviewed. Slated for possible release in 0.9.8 is support for transparent loading of ''dynamic'' ENGINEs (built as self-contained shared−libraries). This would allow ENGINE implementations to be provided independently of OpenSSL libraries and/or OpenSSL-based applications, and would also remove any requirement for applications to explicitly use the ''dynamic'' ENGINE to bind to shared-library implementations.

SEE ALSO

openssl_rsa (3), *openssl_dsa* (3), *openssl_dh* (3), *openssl_rand* (3)

NAME

err − error codes

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/err.h>

unsigned long ERR_get_error(void);
unsigned long ERR_peek_error(void);
unsigned long ERR_get_error_line(const char **file, int *line);
unsigned long ERR_peek_error_line(const char **file, int *line);
unsigned long ERR_get_error_line_data(const char **file, int *line,
        const char **data, int *flags);
unsigned long ERR_peek_error_line_data(const char **file, int *line,
        const char **data, int *flags);

int ERR_GET_LIB(unsigned long e);
int ERR_GET_FUNC(unsigned long e);
int ERR_GET_REASON(unsigned long e);

void ERR_clear_error(void);

char *ERR_error_string(unsigned long e, char *buf);
const char *ERR_lib_error_string(unsigned long e);
const char *ERR_func_error_string(unsigned long e);
const char *ERR_reason_error_string(unsigned long e);

void ERR_print_errors(BIO *bp);
void ERR_print_errors_fp(FILE *fp);

void ERR_load_crypto_strings(void);
void ERR_free_strings(void);

void ERR_remove_state(unsigned long pid);

void ERR_put_error(int lib, int func, int reason, const char *file,
        int line);
void ERR_add_error_data(int num, ...);

void ERR_load_strings(int lib,ERR_STRING_DATA str[]);
unsigned long ERR_PACK(int lib, int func, int reason);
int ERR_get_next_error_library(void);
```

DESCRIPTION

When a call to the OpenSSL library fails, this is usually signalled by the return value, and an error code is stored in an error queue associated with the current thread. The **err** library provides functions to obtain these error codes and textual error messages.

The *ERR_get_error* (3) manpage describes how to access error codes.

Error codes contain information about where the error occurred, and what went wrong. *ERR_GET_LIB* (3) describes how to extract this information. A method to obtain human-readable error messages is described in *ERR_error_string* (3).

ERR_clear_error (3) can be used to clear the error queue.

Note that *ERR_remove_state* (3) should be used to avoid memory leaks when threads are terminated.

ADDING NEW ERROR CODES TO OPENSSL

See *ERR_put_error* (3) if you want to record error codes in the OpenSSL error system from within your application.

The remainder of this section is of interest only if you want to add new error codes to OpenSSL or add

error codes from external libraries.

Reporting errors

Each sub-library has a specific macro *XXXerr()* that is used to report errors. Its first argument is a function code **XXX_F_...**, the second argument is a reason code **XXX_R_...**. Function codes are derived from the function names; reason codes consist of textual error descriptions. For example, the function *ssl23_read()* reports a ''handshake failure'' as follows:

```
SSLerr(SSL_F_SSL23_READ, SSL_R_SSL_HANDSHAKE_FAILURE);
```

Function and reason codes should consist of upper case characters, numbers and underscores only. The error file generation script translates function codes into function names by looking in the header files for an appropriate function name, if none is found it just uses the capitalized form such as ''SSL23_READ'' in the above example.

The trailing section of a reason code (after the ''_R_'') is translated into lower case and underscores changed to spaces.

When you are using new function or reason codes, run **make errors**. The necessary **#define**s will then automatically be added to the sub−library's header file.

Although a library will normally report errors using its own specific XXXerr macro, another library's macro can be used. This is normally only done when a library wants to include ASN1 code which must use the *ASN1err()* macro.

Adding new libraries

When adding a new sub-library to OpenSSL, assign it a library number **ERR_LIB_XXX**, define a macro *XXXerr()* (both in **err.h**), add its name to **ERR_str_libraries[]** (in **crypto/err/err.c**), and add `ERR_load_XXX_strings()` to the *ERR_load_crypto_strings()* function (in **crypto/err/err_all.c**). Finally, add an entry

```
L       XXX     xxx.h   xxx_err.c
```

to **crypto/err/openssl.ec**, and add **xxx_err.c** to the Makefile. Running **make errors** will then generate a file **xxx_err.c**, and add all error codes used in the library to **xxx.h**.

Additionally the library include file must have a certain form. Typically it will initially look like this:

```
#ifndef HEADER_XXX_H
#define HEADER_XXX_H

#ifdef __cplusplus
extern "C" {
#endif

/* Include files */

#include <openssl/bio.h>
#include <openssl/x509.h>

/* Macros, structures and function prototypes */

/* BEGIN ERROR CODES */
```

The **BEGIN ERROR CODES** sequence is used by the error code generation script as the point to place new error codes, any text after this point will be overwritten when **make errors** is run. The closing #endif etc will be automatically added by the script.

The generated C error code file **xxx_err.c** will load the header files **stdio.h**, **openssl/err.h** and **openssl/xxx.h** so the header file must load any additional header files containing any definitions it uses.

USING ERROR CODES IN EXTERNAL LIBRARIES

It is also possible to use OpenSSL's error code scheme in external libraries. The library needs to load its own codes and call the OpenSSL error code insertion script **mkerr.pl** explicitly to add codes to the header

file and generate the C error code file. This will normally be done if the external library needs to generate new ASN1 structures but it can also be used to add more general purpose error code handling.

TBA more details

INTERNALS

The error queues are stored in a hash table with one **ERR_STATE** entry for each pid. *ERR_get_state()* returns the current thread's **ERR_STATE**. An **ERR_STATE** can hold up to **ERR_NUM_ERRORS** error codes. When more error codes are added, the old ones are overwritten, on the assumption that the most recent errors are most important.

Error strings are also stored in hash table. The hash tables can be obtained by calling ERR_get_err_state_table(void) and ERR_get_string_table(void) respectively.

SEE ALSO

CRYPTO_set_id_callback (3), *CRYPTO_set_locking_callback* (3), *ERR_get_error* (3), *ERR_GET_LIB* (3), *ERR_clear_error* (3), *ERR_error_string* (3), *ERR_print_errors* (3), *ERR_load_crypto_strings* (3), *ERR_remove_state* (3), *ERR_put_error* (3), *ERR_load_strings* (3), *SSL_get_error* (3)

NAME

evp − high−level cryptographic functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/evp.h>
```

DESCRIPTION

The EVP library provides a high-level interface to cryptographic functions.

EVP_Seal... and **EVP_Open**... provide public key encryption and decryption to implement digital ‘‘envelopes’’.

The **EVP_Sign**... and **EVP_Verify**... functions implement digital signatures.

Symmetric encryption is available with the **EVP_Encrypt**... functions. The **EVP_Digest**... functions provide message digests.

The **EVP_PKEY**... functions provide a high level interface to asymmetric algorithms.

Algorithms are loaded with *OpenSSL_add_all_algorithms* (3).

All the symmetric algorithms (ciphers), digests and asymmetric algorithms (public key algorithms) can be replaced by ENGINE modules providing alternative implementations. If ENGINE implementations of ciphers or digests are registered as defaults, then the various EVP functions will automatically use those implementations automatically in preference to built in software implementations. For more information, consult the *engine* (3) man page.

Although low level algorithm specific functions exist for many algorithms their use is discouraged. They cannot be used with an ENGINE and ENGINE versions of new algorithms cannot be accessed using the low level functions. Also makes code harder to adapt to new algorithms and some options are not cleanly supported at the low level and some operations are more efficient using the high level interface.

SEE ALSO

EVP_DigestInit (3), *EVP_EncryptInit* (3), *EVP_OpenInit* (3), *EVP_SealInit* (3), *EVP_SignInit* (3), *EVP_VerifyInit* (3), *OpenSSL_add_all_algorithms* (3), *engine* (3)

NAME

HMAC, HMAC_Init, HMAC_Update, HMAC_Final, HMAC_cleanup − HMAC message authentication code

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/hmac.h>

unsigned char *HMAC(const EVP_MD *evp_md, const void *key,
             int key_len, const unsigned char *d, int n,
             unsigned char *md, unsigned int *md_len);

void HMAC_CTX_init(HMAC_CTX *ctx);

void HMAC_Init(HMAC_CTX *ctx, const void *key, int key_len,
             const EVP_MD *md);
void HMAC_Init_ex(HMAC_CTX *ctx, const void *key, int key_len,
                  const EVP_MD *md, ENGINE *impl);
void HMAC_Update(HMAC_CTX *ctx, const unsigned char *data, int len);
void HMAC_Final(HMAC_CTX *ctx, unsigned char *md, unsigned int *len);

void HMAC_CTX_cleanup(HMAC_CTX *ctx);
void HMAC_cleanup(HMAC_CTX *ctx);
```

DESCRIPTION

HMAC is a MAC (message authentication code), i.e. a keyed hash function used for message authentication, which is based on a hash function.

HMAC() computes the message authentication code of the **n** bytes at **d** using the hash function **evp_md** and the key **key** which is **key_len** bytes long.

It places the result in **md** (which must have space for the output of the hash function, which is no more than **EVP_MAX_MD_SIZE** bytes). If **md** is NULL, the digest is placed in a static array. The size of the output is placed in **md_len**, unless it is **NULL**.

evp_md can be *EVP_sha1()*, *EVP_ripemd160()* etc. **key** and **evp_md** may be **NULL** if a key and hash function have been set in a previous call to *HMAC_Init()* for that **HMAC_CTX**.

HMAC_CTX_init() initialises a **HMAC_CTX** before first use. It must be called.

HMAC_CTX_cleanup() erases the key and other data from the **HMAC_CTX** and releases any associated resources. It must be called when an **HMAC_CTX** is no longer required.

HMAC_cleanup() is an alias for *HMAC_CTX_cleanup()* included for back compatibility with 0.9.6b, it is deprecated.

The following functions may be used if the message is not completely stored in memory:

HMAC_Init() initializes a **HMAC_CTX** structure to use the hash function **evp_md** and the key **key** which is **key_len** bytes long. It is deprecated and only included for backward compatibility with OpenSSL 0.9.6b.

HMAC_Init_ex() initializes or reuses a **HMAC_CTX** structure to use the function **evp_md** and key **key**. Either can be NULL, in which case the existing one will be reused. *HMAC_CTX_init()* must have been called before the first use of an **HMAC_CTX** in this function. **N.B. *HMAC_Init()* had this undocumented behaviour in previous versions of OpenSSL − failure to switch to *HMAC_Init_ex()* in programs that expect it will cause them to stop working**.

HMAC_Update() can be called repeatedly with chunks of the message to be authenticated (**len** bytes at **data**).

HMAC_Final() places the message authentication code in **md**, which must have space for the hash function output.

RETURN VALUES

HMAC() returns a pointer to the message authentication code.

HMAC_CTX_init(), *HMAC_Init_ex()*, *HMAC_Update()*, *HMAC_Final()* and *HMAC_CTX_cleanup()* do not return values.

CONFORMING TO

RFC 2104

SEE ALSO

openssl_sha (3), *openssl_evp* (3)

HISTORY

HMAC(), *HMAC_Init()*, *HMAC_Update()*, *HMAC_Final()* and *HMAC_cleanup()* are available since SSLeay 0.9.0.

HMAC_CTX_init(), *HMAC_Init_ex()* and *HMAC_CTX_cleanup()* are available since OpenSSL 0.9.7.

NAME

lh_new, lh_free, lh_insert, lh_delete, lh_retrieve, lh_doall, lh_doall_arg, lh_error − dynamic hash table

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/lhash.h>

LHASH *lh_new(LHASH_HASH_FN_TYPE hash, LHASH_COMP_FN_TYPE compare);
void lh_free(LHASH *table);

void *lh_insert(LHASH *table, void *data);
void *lh_delete(LHASH *table, void *data);
void *lh_retrieve(LHASH *table, void *data);

void lh_doall(LHASH *table, LHASH_DOALL_FN_TYPE func);
void lh_doall_arg(LHASH *table, LHASH_DOALL_ARG_FN_TYPE func,
         void *arg);

int lh_error(LHASH *table);

typedef int (*LHASH_COMP_FN_TYPE)(const void *, const void *);
typedef unsigned long (*LHASH_HASH_FN_TYPE)(const void *);
typedef void (*LHASH_DOALL_FN_TYPE)(const void *);
typedef void (*LHASH_DOALL_ARG_FN_TYPE)(const void *, const void *);
```

DESCRIPTION

This library implements dynamic hash tables. The hash table entries can be arbitrary structures. Usually they consist of key and value fields.

lh_new() creates a new **LHASH** structure to store arbitrary data entries, and provides the 'hash' and 'compare' callbacks to be used in organising the table's entries. The **hash** callback takes a pointer to a table entry as its argument and returns an unsigned long hash value for its key field. The hash value is normally truncated to a power of 2, so make sure that your hash function returns well mixed low order bits. The **compare** callback takes two arguments (pointers to two hash table entries), and returns 0 if their keys are equal, non-zero otherwise. If your hash table will contain items of some particular type and the **hash** and **compare** callbacks hash/compare these types, then the **DECLARE_LHASH_HASH_FN** and **IMPLEMENT_LHASH_COMP_FN** macros can be used to create callback wrappers of the prototypes required by *lh_new()*. These provide per-variable casts before calling the type-specific callbacks written by the application author. These macros, as well as those used for the ''doall'' callbacks, are defined as;

```
#define DECLARE_LHASH_HASH_FN(f_name,o_type) \
        unsigned long f_name##_LHASH_HASH(const void *);
#define IMPLEMENT_LHASH_HASH_FN(f_name,o_type) \
        unsigned long f_name##_LHASH_HASH(const void *arg) { \
                o_type a = (o_type)arg; \
                return f_name(a); }
#define LHASH_HASH_FN(f_name) f_name##_LHASH_HASH

#define DECLARE_LHASH_COMP_FN(f_name,o_type) \
        int f_name##_LHASH_COMP(const void *, const void *);
#define IMPLEMENT_LHASH_COMP_FN(f_name,o_type) \
        int f_name##_LHASH_COMP(const void *arg1, const void *arg2) { \
                o_type a = (o_type)arg1; \
                o_type b = (o_type)arg2; \
                return f_name(a,b); }
#define LHASH_COMP_FN(f_name) f_name##_LHASH_COMP
```

```
#define DECLARE_LHASH_DOALL_FN(f_name,o_type) \
        void f_name##_LHASH_DOALL(const void *);
#define IMPLEMENT_LHASH_DOALL_FN(f_name,o_type) \
        void f_name##_LHASH_DOALL(const void *arg) { \
                o_type a = (o_type)arg; \
                f_name(a); }
#define LHASH_DOALL_FN(f_name) f_name##_LHASH_DOALL

#define DECLARE_LHASH_DOALL_ARG_FN(f_name,o_type,a_type) \
        void f_name##_LHASH_DOALL_ARG(const void *, const void *);
#define IMPLEMENT_LHASH_DOALL_ARG_FN(f_name,o_type,a_type) \
        void f_name##_LHASH_DOALL_ARG(const void *arg1, const void *arg2) { \
                o_type a = (o_type)arg1; \
                a_type b = (a_type)arg2; \
                f_name(a,b); }
#define LHASH_DOALL_ARG_FN(f_name) f_name##_LHASH_DOALL_ARG
```

An example of a hash table storing (pointers to) structures of type 'STUFF' could be defined as follows;

```
/* Calculates the hash value of 'tohash' (implemented elsewhere) */
unsigned long STUFF_hash(const STUFF *tohash);
/* Orders 'arg1' and 'arg2' (implemented elsewhere) */
int STUFF_cmp(const STUFF *arg1, const STUFF *arg2);
/* Create the type-safe wrapper functions for use in the LHASH internals */
static IMPLEMENT_LHASH_HASH_FN(STUFF_hash, const STUFF *)
static IMPLEMENT_LHASH_COMP_FN(STUFF_cmp, const STUFF *);
/* ... */
int main(int argc, char *argv[]) {
        /* Create the new hash table using the hash/compare wrappers */
        LHASH *hashtable = lh_new(LHASH_HASH_FN(STUFF_hash),
                                  LHASH_COMP_FN(STUFF_cmp));
        /* ... */
}
```

lh_free() frees the **LHASH** structure **table**. Allocated hash table entries will not be freed; consider using *lh_doall()* to deallocate any remaining entries in the hash table (see below).

lh_insert() inserts the structure pointed to by **data** into **table**. If there already is an entry with the same key, the old value is replaced. Note that *lh_insert()* stores pointers, the data are not copied.

lh_delete() deletes an entry from **table**.

lh_retrieve() looks up an entry in **table**. Normally, **data** is a structure with the key field(s) set; the function will return a pointer to a fully populated structure.

lh_doall() will, for every entry in the hash table, call **func** with the data item as its parameter. For *lh_doall()* and *lh_doall_arg()*, function pointer casting should be avoided in the callbacks (see **NOTE**) − instead, either declare the callbacks to match the prototype required in *lh_new()* or use the declare/implement macros to create type-safe wrappers that cast variables prior to calling your type-specific callbacks. An example of this is illustrated here where the callback is used to cleanup resources for items in the hash table prior to the hashtable itself being deallocated:

```
/* Cleans up resources belonging to 'a' (this is implemented elsewhere) */
void STUFF_cleanup(STUFF *a);
/* Implement a prototype-compatible wrapper for "STUFF_cleanup" */
IMPLEMENT_LHASH_DOALL_FN(STUFF_cleanup, STUFF *)
        /* ... then later in the code ... */
/* So to run "STUFF_cleanup" against all items in a hash table ... */
lh_doall(hashtable, LHASH_DOALL_FN(STUFF_cleanup));
/* Then the hash table itself can be deallocated */
lh_free(hashtable);
```

When doing this, be careful if you delete entries from the hash table in your callbacks: the table may decrease in size, moving the item that you are currently on down lower in the hash table − this could cause some entries to be skipped during the iteration. The second best solution to this problem is to set hash−>down_load=0 before you start (which will stop the hash table ever decreasing in size). The best solution is probably to avoid deleting items from the hash table inside a ''doall'' callback!

lh_doall_arg() is the same as *lh_doall()* except that **func** will be called with **arg** as the second argument and **func** should be of type **LHASH_DOALL_ARG_FN_TYPE** (a callback prototype that is passed both the table entry and an extra argument). As with *lh_doall()*, you can instead choose to declare your callback with a prototype matching the types you are dealing with and use the declare/implement macros to create compatible wrappers that cast variables before calling your type-specific callbacks. An example of this is demonstrated here (printing all hash table entries to a BIO that is provided by the caller):

```
/* Prints item 'a' to 'output_bio' (this is implemented elsewhere) */
void STUFF_print(const STUFF *a, BIO *output_bio);
/* Implement a prototype-compatible wrapper for "STUFF_print" */
static IMPLEMENT_LHASH_DOALL_ARG_FN(STUFF_print, const STUFF *, BIO *)
        /* ... then later in the code ... */
/* Print out the entire hashtable to a particular BIO */
lh_doall_arg(hashtable, LHASH_DOALL_ARG_FN(STUFF_print), logging_bio);
```

lh_error() can be used to determine if an error occurred in the last operation. *lh_error()* is a macro.

RETURN VALUES

lh_new() returns **NULL** on error, otherwise a pointer to the new **LHASH** structure.

When a hash table entry is replaced, *lh_insert()* returns the value being replaced. **NULL** is returned on normal operation and on error.

lh_delete() returns the entry being deleted. **NULL** is returned if there is no such value in the hash table.

lh_retrieve() returns the hash table entry if it has been found, **NULL** otherwise.

lh_error() returns 1 if an error occurred in the last operation, 0 otherwise.

lh_free(), *lh_doall()* and *lh_doall_arg()* return no values.

NOTE

The various LHASH macros and callback types exist to make it possible to write type-safe code without resorting to function-prototype casting − an evil that makes application code much harder to audit/verify and also opens the window of opportunity for stack corruption and other hard-to-find bugs. It also, apparently, violates ANSI−C.

The LHASH code regards table entries as constant data. As such, it internally represents *lh_insert()*'d items with a ''const void *'' pointer type. This is why callbacks such as those used by *lh_doall()* and *lh_doall_arg()* declare their prototypes with ''const'', even for the parameters that pass back the table items' data pointers − for consistency, user-provided data is ''const'' at all times as far as the LHASH code is concerned. However, as callers are themselves providing these pointers, they can choose whether they too should be treating all such parameters as constant.

As an example, a hash table may be maintained by code that, for reasons of encapsulation, has only ''const'' access to the data being indexed in the hash table (ie. it is returned as ''const'' from elsewhere in

their code) − in this case the LHASH prototypes are appropriate as−is. Conversely, if the caller is responsible for the life-time of the data in question, then they may well wish to make modifications to table item passed back in the *lh_doall()* or *lh_doall_arg()* callbacks (see the ''STUFF_cleanup'' example above). If so, the caller can either cast the ''const'' away (if they're providing the raw callbacks themselves) or use the macros to declare/implement the wrapper functions without ''const'' types.

Callers that only have ''const'' access to data they're indexing in a table, yet declare callbacks without constant types (or cast the ''const'' away themselves), are therefore creating their own risks/bugs without being encouraged to do so by the API. On a related note, those auditing code should pay special attention to any instances of DECLARE/IMPLEMENT_LHASH_DOALL_[ARG_]_FN macros that provide types without any ''const'' qualifiers.

BUGS

lh_insert() returns **NULL** both for success and error.

INTERNALS

The following description is based on the SSLeay documentation:

The **lhash** library implements a hash table described in the *Communications of the ACM* in 1991. What makes this hash table different is that as the table fills, the hash table is increased (or decreased) in size via *OPENSSL_realloc()*. When a 'resize' is done, instead of all hashes being redistributed over twice as many 'buckets', one bucket is split. So when an 'expand' is done, there is only a minimal cost to redistribute some values. Subsequent inserts will cause more single 'bucket' redistributions but there will never be a sudden large cost due to redistributing all the 'buckets'.

The state for a particular hash table is kept in the **LHASH** structure. The decision to increase or decrease the hash table size is made depending on the 'load' of the hash table. The load is the number of items in the hash table divided by the size of the hash table. The default values are as follows. If (hash−>up_load < load) => expand. if (hash−>down_load > load) => contract. The **up_load** has a default value of 1 and **down_load** has a default value of 2. These numbers can be modified by the application by just playing with the **up_load** and **down_load** variables. The 'load' is kept in a form which is multiplied by 256. So hash−>up_load=8*256; will cause a load of 8 to be set.

If you are interested in performance the field to watch is num_comp_calls. The hash library keeps track of the 'hash' value for each item so when a lookup is done, the 'hashes' are compared, if there is a match, then a full compare is done, and hash−>num_comp_calls is incremented. If num_comp_calls is not equal to num_delete plus num_retrieve it means that your hash function is generating hashes that are the same for different values. It is probably worth changing your hash function if this is the case because even if your hash table has 10 items in a 'bucket', it can be searched with 10 **unsigned long** compares and 10 linked list traverses. This will be much less expensive that 10 calls to your compare function.

lh_strhash() is a demo string hashing function:

```
unsigned long lh_strhash(const char *c);
```

Since the **LHASH** routines would normally be passed structures, this routine would not normally be passed to *lh_new()*, rather it would be used in the function passed to *lh_new()*.

SEE ALSO

lh_stats (3)

HISTORY

The **lhash** library is available in all versions of SSLeay and OpenSSL. *lh_error()* was added in SSLeay 0.9.1b.

This manpage is derived from the SSLeay documentation.

In OpenSSL 0.9.7, all lhash functions that were passed function pointers were changed for better type safety, and the function types LHASH_COMP_FN_TYPE, LHASH_HASH_FN_TYPE, LHASH_DOALL_FN_TYPE and LHASH_DOALL_ARG_FN_TYPE became available.

NAME

MD2, MD4, MD5, MD2_Init, MD2_Update, MD2_Final, MD4_Init, MD4_Update, MD4_Final, MD5_Init, MD5_Update, MD5_Final − MD2, MD4, and MD5 hash functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/md2.h>

unsigned char *MD2(const unsigned char *d, unsigned long n,
                   unsigned char *md);

int MD2_Init(MD2_CTX *c);
int MD2_Update(MD2_CTX *c, const unsigned char *data,
                   unsigned long len);
int MD2_Final(unsigned char *md, MD2_CTX *c);

#include <openssl/md4.h>

unsigned char *MD4(const unsigned char *d, unsigned long n,
                   unsigned char *md);

int MD4_Init(MD4_CTX *c);
int MD4_Update(MD4_CTX *c, const void *data,
                   unsigned long len);
int MD4_Final(unsigned char *md, MD4_CTX *c);

#include <openssl/md5.h>

unsigned char *MD5(const unsigned char *d, unsigned long n,
                   unsigned char *md);

int MD5_Init(MD5_CTX *c);
int MD5_Update(MD5_CTX *c, const void *data,
                   unsigned long len);
int MD5_Final(unsigned char *md, MD5_CTX *c);
```

DESCRIPTION

MD2, MD4, and MD5 are cryptographic hash functions with a 128 bit output.

MD2(), *MD4()*, and *MD5()* compute the MD2, MD4, and MD5 message digest of the **n** bytes at **d** and place it in **md** (which must have space for MD2_DIGEST_LENGTH == MD4_DIGEST_LENGTH == MD5_DIGEST_LENGTH == 16 bytes of output). If **md** is NULL, the digest is placed in a static array.

The following functions may be used if the message is not completely stored in memory:

MD2_Init() initializes a **MD2_CTX** structure.

MD2_Update() can be called repeatedly with chunks of the message to be hashed (**len** bytes at **data**).

MD2_Final() places the message digest in **md**, which must have space for MD2_DIGEST_LENGTH == 16 bytes of output, and erases the **MD2_CTX**.

MD4_Init(), *MD4_Update()*, *MD4_Final()*, *MD5_Init()*, *MD5_Update()*, and *MD5_Final()* are analogous using an **MD4_CTX** and **MD5_CTX** structure.

Applications should use the higher level functions *EVP_DigestInit*(3) etc. instead of calling the hash functions directly.

NOTE

MD2, MD4, and MD5 are recommended only for compatibility with existing applications. In new applications, SHA−1 or RIPEMD−160 should be preferred.

RETURN VALUES

MD2(), *MD4()*, and *MD5()* return pointers to the hash value.

MD2_Init(), *MD2_Update()*, *MD2_Final()*, *MD4_Init()*, *MD4_Update()*, *MD4_Final()*, *MD5_Init()*, *MD5_Update()*, and *MD5_Final()* return 1 for success, 0 otherwise.

CONFORMING TO

RFC 1319, RFC 1320, RFC 1321

SEE ALSO

openssl_sha (3), *openssl_ripemd* (3), *EVP_DigestInit* (3)

HISTORY

MD2(), *MD2_Init()*, *MD2_Update()* *MD2_Final()*, *MD5()*, *MD5_Init()*, *MD5_Update()* and *MD5_Final()* are available in all versions of SSLeay and OpenSSL.

MD4(), *MD4_Init()*, and *MD4_Update()* are available in OpenSSL 0.9.6 and above.

NAME

MDC2, MDC2_Init, MDC2_Update, MDC2_Final − MDC2 hash function

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/mdc2.h>

unsigned char *MDC2(const unsigned char *d, unsigned long n,
                    unsigned char *md);

int MDC2_Init(MDC2_CTX *c);
int MDC2_Update(MDC2_CTX *c, const unsigned char *data,
                    unsigned long len);
int MDC2_Final(unsigned char *md, MDC2_CTX *c);
```

DESCRIPTION

MDC2 is a method to construct hash functions with 128 bit output from block ciphers. These functions are an implementation of MDC2 with DES.

MDC2() computes the MDC2 message digest of the **n** bytes at **d** and places it in **md** (which must have space for MDC2_DIGEST_LENGTH == 16 bytes of output). If **md** is NULL, the digest is placed in a static array.

The following functions may be used if the message is not completely stored in memory:

MDC2_Init() initializes a **MDC2_CTX** structure.

MDC2_Update() can be called repeatedly with chunks of the message to be hashed (**len** bytes at **data**).

MDC2_Final() places the message digest in **md**, which must have space for MDC2_DIGEST_LENGTH == 16 bytes of output, and erases the **MDC2_CTX**.

Applications should use the higher level functions *EVP_DigestInit*(3) etc. instead of calling the hash functions directly.

RETURN VALUES

MDC2() returns a pointer to the hash value.

MDC2_Init(), *MDC2_Update()* and *MDC2_Final()* return 1 for success, 0 otherwise.

CONFORMING TO

ISO/IEC 10118−2, with DES

SEE ALSO

openssl_sha(3), *EVP_DigestInit*(3)

HISTORY

MDC2(), *MDC2_Init()*, *MDC2_Update()* and *MDC2_Final()* are available since SSLeay 0.8.

NAME

PEM − PEM routines

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/pem.h>

EVP_PKEY *PEM_read_bio_PrivateKey(BIO *bp, EVP_PKEY **x,
                                  pem_password_cb *cb, void *u);

EVP_PKEY *PEM_read_PrivateKey(FILE *fp, EVP_PKEY **x,
                              pem_password_cb *cb, void *u);

int PEM_write_bio_PrivateKey(BIO *bp, EVP_PKEY *x, const EVP_CIPHER *enc,
                             unsigned char *kstr, int klen,
                             pem_password_cb *cb, void *u);

int PEM_write_PrivateKey(FILE *fp, EVP_PKEY *x, const EVP_CIPHER *enc,
                         unsigned char *kstr, int klen,
                         pem_password_cb *cb, void *u);

int PEM_write_bio_PKCS8PrivateKey(BIO *bp, EVP_PKEY *x, const EVP_CIPHER *enc,
                                  char *kstr, int klen,
                                  pem_password_cb *cb, void *u);

int PEM_write_PKCS8PrivateKey(FILE *fp, EVP_PKEY *x, const EVP_CIPHER *enc,
                              char *kstr, int klen,
                              pem_password_cb *cb, void *u);

int PEM_write_bio_PKCS8PrivateKey_nid(BIO *bp, EVP_PKEY *x, int nid,
                                      char *kstr, int klen,
                                      pem_password_cb *cb, void *u);

int PEM_write_PKCS8PrivateKey_nid(FILE *fp, EVP_PKEY *x, int nid,
                                  char *kstr, int klen,
                                  pem_password_cb *cb, void *u);

EVP_PKEY *PEM_read_bio_PUBKEY(BIO *bp, EVP_PKEY **x,
                              pem_password_cb *cb, void *u);

EVP_PKEY *PEM_read_PUBKEY(FILE *fp, EVP_PKEY **x,
                          pem_password_cb *cb, void *u);

int PEM_write_bio_PUBKEY(BIO *bp, EVP_PKEY *x);
int PEM_write_PUBKEY(FILE *fp, EVP_PKEY *x);

RSA *PEM_read_bio_RSAPrivateKey(BIO *bp, RSA **x,
                                pem_password_cb *cb, void *u);

RSA *PEM_read_RSAPrivateKey(FILE *fp, RSA **x,
                            pem_password_cb *cb, void *u);

int PEM_write_bio_RSAPrivateKey(BIO *bp, RSA *x, const EVP_CIPHER *enc,
                                unsigned char *kstr, int klen,
                                pem_password_cb *cb, void *u);

int PEM_write_RSAPrivateKey(FILE *fp, RSA *x, const EVP_CIPHER *enc,
                            unsigned char *kstr, int klen,
                            pem_password_cb *cb, void *u);

RSA *PEM_read_bio_RSAPublicKey(BIO *bp, RSA **x,
                               pem_password_cb *cb, void *u);
```

```
RSA *PEM_read_RSAPublicKey(FILE *fp, RSA **x,
                                        pem_password_cb *cb, void *u);
int PEM_write_bio_RSAPublicKey(BIO *bp, RSA *x);
int PEM_write_RSAPublicKey(FILE *fp, RSA *x);
RSA *PEM_read_bio_RSA_PUBKEY(BIO *bp, RSA **x,
                                        pem_password_cb *cb, void *u);
RSA *PEM_read_RSA_PUBKEY(FILE *fp, RSA **x,
                                        pem_password_cb *cb, void *u);
int PEM_write_bio_RSA_PUBKEY(BIO *bp, RSA *x);
int PEM_write_RSA_PUBKEY(FILE *fp, RSA *x);
DSA *PEM_read_bio_DSAPrivateKey(BIO *bp, DSA **x,
                                        pem_password_cb *cb, void *u);
DSA *PEM_read_DSAPrivateKey(FILE *fp, DSA **x,
                                        pem_password_cb *cb, void *u);
int PEM_write_bio_DSAPrivateKey(BIO *bp, DSA *x, const EVP_CIPHER *enc,
                                        unsigned char *kstr, int klen,
                                        pem_password_cb *cb, void *u);
int PEM_write_DSAPrivateKey(FILE *fp, DSA *x, const EVP_CIPHER *enc,
                                        unsigned char *kstr, int klen,
                                        pem_password_cb *cb, void *u);
DSA *PEM_read_bio_DSA_PUBKEY(BIO *bp, DSA **x,
                                        pem_password_cb *cb, void *u);
DSA *PEM_read_DSA_PUBKEY(FILE *fp, DSA **x,
                                        pem_password_cb *cb, void *u);
int PEM_write_bio_DSA_PUBKEY(BIO *bp, DSA *x);
int PEM_write_DSA_PUBKEY(FILE *fp, DSA *x);
DSA *PEM_read_bio_DSAparams(BIO *bp, DSA **x, pem_password_cb *cb, void *u);
DSA *PEM_read_DSAparams(FILE *fp, DSA **x, pem_password_cb *cb, void *u);
int PEM_write_bio_DSAparams(BIO *bp, DSA *x);
int PEM_write_DSAparams(FILE *fp, DSA *x);
DH *PEM_read_bio_DHparams(BIO *bp, DH **x, pem_password_cb *cb, void *u);
DH *PEM_read_DHparams(FILE *fp, DH **x, pem_password_cb *cb, void *u);
int PEM_write_bio_DHparams(BIO *bp, DH *x);
int PEM_write_DHparams(FILE *fp, DH *x);
X509 *PEM_read_bio_X509(BIO *bp, X509 **x, pem_password_cb *cb, void *u);
X509 *PEM_read_X509(FILE *fp, X509 **x, pem_password_cb *cb, void *u);
int PEM_write_bio_X509(BIO *bp, X509 *x);
int PEM_write_X509(FILE *fp, X509 *x);
X509 *PEM_read_bio_X509_AUX(BIO *bp, X509 **x, pem_password_cb *cb, void *u);
X509 *PEM_read_X509_AUX(FILE *fp, X509 **x, pem_password_cb *cb, void *u);
int PEM_write_bio_X509_AUX(BIO *bp, X509 *x);
int PEM_write_X509_AUX(FILE *fp, X509 *x);
```

```
X509_REQ *PEM_read_bio_X509_REQ(BIO *bp, X509_REQ **x,
                                pem_password_cb *cb, void *u);
X509_REQ *PEM_read_X509_REQ(FILE *fp, X509_REQ **x,
                                pem_password_cb *cb, void *u);
int PEM_write_bio_X509_REQ(BIO *bp, X509_REQ *x);
int PEM_write_X509_REQ(FILE *fp, X509_REQ *x);
int PEM_write_bio_X509_REQ_NEW(BIO *bp, X509_REQ *x);
int PEM_write_X509_REQ_NEW(FILE *fp, X509_REQ *x);
X509_CRL *PEM_read_bio_X509_CRL(BIO *bp, X509_CRL **x,
                                pem_password_cb *cb, void *u);
X509_CRL *PEM_read_X509_CRL(FILE *fp, X509_CRL **x,
                                pem_password_cb *cb, void *u);
int PEM_write_bio_X509_CRL(BIO *bp, X509_CRL *x);
int PEM_write_X509_CRL(FILE *fp, X509_CRL *x);
PKCS7 *PEM_read_bio_PKCS7(BIO *bp, PKCS7 **x, pem_password_cb *cb, void *u);
PKCS7 *PEM_read_PKCS7(FILE *fp, PKCS7 **x, pem_password_cb *cb, void *u);
int PEM_write_bio_PKCS7(BIO *bp, PKCS7 *x);
int PEM_write_PKCS7(FILE *fp, PKCS7 *x);
NETSCAPE_CERT_SEQUENCE *PEM_read_bio_NETSCAPE_CERT_SEQUENCE(BIO *bp,
                                        NETSCAPE_CERT_SEQUENCE **x,
                                        pem_password_cb *cb, void *u);
NETSCAPE_CERT_SEQUENCE *PEM_read_NETSCAPE_CERT_SEQUENCE(FILE *fp,
                                        NETSCAPE_CERT_SEQUENCE **x,
                                        pem_password_cb *cb, void *u);
int PEM_write_bio_NETSCAPE_CERT_SEQUENCE(BIO *bp, NETSCAPE_CERT_SEQUENCE *x);
int PEM_write_NETSCAPE_CERT_SEQUENCE(FILE *fp, NETSCAPE_CERT_SEQUENCE *x);
```

DESCRIPTION

The PEM functions read or write structures in PEM format. In this sense PEM format is simply base64 encoded data surrounded by header lines.

For more details about the meaning of arguments see the **PEM FUNCTION ARGUMENTS** section.

Each operation has four functions associated with it. For clarity the term "**foobar** functions" will be used to collectively refer to the *PEM_read_bio_foobar()*, *PEM_read_foobar()*, *PEM_write_bio_foobar()* and *PEM_write_foobar()* functions.

The **PrivateKey** functions read or write a private key in PEM format using an EVP_PKEY structure. The write routines use ‘‘traditional’’ private key format and can handle both RSA and DSA private keys. The read functions can additionally transparently handle PKCS#8 format encrypted and unencrypted keys too.

PEM_write_bio_PKCS8PrivateKey() and *PEM_write_PKCS8PrivateKey()* write a private key in an EVP_PKEY structure in PKCS#8 EncryptedPrivateKeyInfo format using PKCS#5 v2.0 password based encryption algorithms. The **cipher** argument specifies the encryption algoritm to use: unlike all other PEM routines the encryption is applied at the PKCS#8 level and not in the PEM headers. If **cipher** is NULL then no encryption is used and a PKCS#8 PrivateKeyInfo structure is used instead.

PEM_write_bio_PKCS8PrivateKey_nid() and *PEM_write_PKCS8PrivateKey_nid()* also write out a private key as a PKCS#8 EncryptedPrivateKeyInfo however it uses PKCS#5 v1.5 or PKCS#12 encryption algorithms instead. The algorithm to use is specified in the **nid** parameter and should be the NID of the corresponding OBJECT IDENTIFIER (see NOTES section).

The **PUBKEY** functions process a public key using an EVP_PKEY structure. The public key is encoded as a

SubjectPublicKeyInfo structure.

The **RSAPrivateKey** functions process an RSA private key using an RSA structure. It handles the same formats as the **PrivateKey** functions but an error occurs if the private key is not RSA.

The **RSAPublicKey** functions process an RSA public key using an RSA structure. The public key is encoded using a PKCS#1 RSAPublicKey structure.

The **RSA_PUBKEY** functions also process an RSA public key using an RSA structure. However the public key is encoded using a SubjectPublicKeyInfo structure and an error occurs if the public key is not RSA.

The **DSAPrivateKey** functions process a DSA private key using a DSA structure. It handles the same formats as the **PrivateKey** functions but an error occurs if the private key is not DSA.

The **DSA_PUBKEY** functions process a DSA public key using a DSA structure. The public key is encoded using a SubjectPublicKeyInfo structure and an error occurs if the public key is not DSA.

The **DSAparams** functions process DSA parameters using a DSA structure. The parameters are encoded using a foobar structure.

The **DHparams** functions process DH parameters using a DH structure. The parameters are encoded using a PKCS#3 DHparameter structure.

The **X509** functions process an X509 certificate using an X509 structure. They will also process a trusted X509 certificate but any trust settings are discarded.

The **X509_AUX** functions process a trusted X509 certificate using an X509 structure.

The **X509_REQ** and **X509_REQ_NEW** functions process a PKCS#10 certificate request using an X509_REQ structure. The **X509_REQ** write functions use **CERTIFICATE REQUEST** in the header whereas the **X509_REQ_NEW** functions use **NEW CERTIFICATE REQUEST** (as required by some CAs). The **X509_REQ** read functions will handle either form so there are no **X509_REQ_NEW** read functions.

The **X509_CRL** functions process an X509 CRL using an X509_CRL structure.

The **PKCS7** functions process a PKCS#7 ContentInfo using a PKCS7 structure.

The **NETSCAPE_CERT_SEQUENCE** functions process a Netscape Certificate Sequence using a NETSCAPE_CERT_SEQUENCE structure.

PEM FUNCTION ARGUMENTS

The PEM functions have many common arguments.

The **bp** BIO parameter (if present) specifies the BIO to read from or write to.

The **fp** FILE parameter (if present) specifies the FILE pointer to read from or write to.

The PEM read functions all take an argument **TYPE **x** and return a **TYPE *** pointer. Where **TYPE** is whatever structure the function uses. If **x** is NULL then the parameter is ignored. If **x** is not NULL but ***x** is NULL then the structure returned will be written to ***x**. If neither **x** nor ***x** is NULL then an attempt is made to reuse the structure at ***x** (but see BUGS and EXAMPLES sections). Irrespective of the value of **x** a pointer to the structure is always returned (or NULL if an error occurred).

The PEM functions which write private keys take an **enc** parameter which specifies the encryption algorithm to use, encryption is done at the PEM level. If this parameter is set to NULL then the private key is written in unencrypted form.

The **cb** argument is the callback to use when querying for the pass phrase used for encrypted PEM structures (normally only private keys).

For the PEM write routines if the **kstr** parameter is not NULL then **klen** bytes at **kstr** are used as the passphrase and **cb** is ignored.

If the **cb** parameters is set to NULL and the **u** parameter is not NULL then the **u** parameter is interpreted as a null terminated string to use as the passphrase. If both **cb** and **u** are NULL then the default callback routine is used which will typically prompt for the passphrase on the current terminal with echoing turned off.

The default passphrase callback is sometimes inappropriate (for example in a GUI application) so an

alternative can be supplied. The callback routine has the following form:

```
int cb(char *buf, int size, int rwflag, void *u);
```

buf is the buffer to write the passphrase to. **size** is the maximum length of the passphrase (i.e. the size of buf). **rwflag** is a flag which is set to 0 when reading and 1 when writing. A typical routine will ask the user to verify the passphrase (for example by prompting for it twice) if **rwflag** is 1. The **u** parameter has the same value as the **u** parameter passed to the PEM routine. It allows arbitrary data to be passed to the callback by the application (for example a window handle in a GUI application). The callback **must** return the number of characters in the passphrase or 0 if an error occurred.

EXAMPLES

Although the PEM routines take several arguments in almost all applications most of them are set to 0 or NULL.

Read a certificate in PEM format from a BIO:

```
X509 *x;
x = PEM_read_bio_X509(bp, NULL, 0, NULL);
if (x == NULL)
       {
       /* Error */
       }
```

Alternative method:

```
X509 *x = NULL;
if (!PEM_read_bio_X509(bp, &x, 0, NULL))
       {
       /* Error */
       }
```

Write a certificate to a BIO:

```
if (!PEM_write_bio_X509(bp, x))
       {
       /* Error */
       }
```

Write an unencrypted private key to a FILE pointer:

```
if (!PEM_write_PrivateKey(fp, key, NULL, NULL, 0, 0, NULL))
       {
       /* Error */
       }
```

Write a private key (using traditional format) to a BIO using triple DES encryption, the pass phrase is prompted for:

```
if (!PEM_write_bio_PrivateKey(bp, key, EVP_des_ede3_cbc(), NULL, 0, 0, NULL))
       {
       /* Error */
       }
```

Write a private key (using PKCS#8 format) to a BIO using triple DES encryption, using the pass phrase ‘‘hello’’:

```
if (!PEM_write_bio_PKCS8PrivateKey(bp, key, EVP_des_ede3_cbc(), NULL, 0, 0, "hello"
       {
       /* Error */
       }
```

Read a private key from a BIO using the pass phrase ‘‘hello’’:

```
key = PEM_read_bio_PrivateKey(bp, NULL, 0, "hello");
if (key == NULL)
       {
       /* Error */
       }
```

Read a private key from a BIO using a pass phrase callback:

```
key = PEM_read_bio_PrivateKey(bp, NULL, pass_cb, "My Private Key");
if (key == NULL)
       {
       /* Error */
       }
```

Skeleton pass phrase callback:

```
int pass_cb(char *buf, int size, int rwflag, void *u);
       {
       int len;
       char *tmp;
       /* We'd probably do something else if 'rwflag' is 1 */
       printf("Enter pass phrase for \"%s\"\n", u);

       /* get pass phrase, length 'len' into 'tmp' */
       tmp = "hello";
       len = strlen(tmp);

       if (len <= 0) return 0;
       /* if too long, truncate */
       if (len > size) len = size;
       memcpy(buf, tmp, len);
       return len;
       }
```

NOTES

The old **PrivateKey** write routines are retained for compatibility. New applications should write private keys using the *PEM_write_bio_PKCS8PrivateKey()* or *PEM_write_PKCS8PrivateKey()* routines because they are more secure (they use an iteration count of 2048 whereas the traditional routines use a count of 1) unless compatibility with older versions of OpenSSL is important.

The **PrivateKey** read routines can be used in all applications because they handle all formats transparently.

A frequent cause of problems is attempting to use the PEM routines like this:

```
X509 *x;
PEM_read_bio_X509(bp, &x, 0, NULL);
```

this is a bug because an attempt will be made to reuse the data at **x** which is an uninitialised pointer.

PEM ENCRYPTION FORMAT

This old **PrivateKey** routines use a non standard technique for encryption.

The private key (or other data) takes the following form:

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,3F17F5316E2BAC89

...base64 encoded data...
-----END RSA PRIVATE KEY-----
```

The line beginning DEK-Info contains two comma separated pieces of information: the encryption algorithm name as used by *EVP_get_cipherbyname()* and an 8 byte **salt** encoded as a set of hexadecimal digits.

After this is the base64 encoded encrypted data.

The encryption key is determined using *EVP_bytestokey()*, using **salt** and an iteration count of 1. The IV used is the value of **salt** and *not* the IV returned by *EVP_bytestokey()*.

BUGS

The PEM read routines in some versions of OpenSSL will not correctly reuse an existing structure. Therefore the following:

```
PEM_read_bio_X509(bp, &x, 0, NULL);
```

where **x** already contains a valid certificate, may not work, whereas:

```
X509_free(x);
x = PEM_read_bio_X509(bp, NULL, 0, NULL);
```

is guaranteed to work.

RETURN CODES

The read routines return either a pointer to the structure read or NULL if an error occurred.

The write routines return 1 for success or 0 for failure.

NAME

rand − pseudo−random number generator

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rand.h>

int  RAND_set_rand_engine(ENGINE *engine);

int  RAND_bytes(unsigned char *buf, int num);
int  RAND_pseudo_bytes(unsigned char *buf, int num);

void RAND_seed(const void *buf, int num);
void RAND_add(const void *buf, int num, int entropy);
int  RAND_status(void);

int  RAND_load_file(const char *file, long max_bytes);
int  RAND_write_file(const char *file);
const char *RAND_file_name(char *file, size_t num);

int  RAND_egd(const char *path);

void RAND_set_rand_method(const RAND_METHOD *meth);
const RAND_METHOD *RAND_get_rand_method(void);
RAND_METHOD *RAND_SSLeay(void);

void RAND_cleanup(void);

/* For Win32 only */
void RAND_screen(void);
int RAND_event(UINT, WPARAM, LPARAM);
```

DESCRIPTION

Since the introduction of the ENGINE API, the recommended way of controlling default implementations is by using the ENGINE API functions. The default **RAND_METHOD**, as set by *RAND_set_rand_method()* and returned by *RAND_get_rand_method()*, is only used if no ENGINE has been set as the default ‘‘rand’’ implementation. Hence, these two functions are no longer the recommened way to control defaults.

If an alternative **RAND_METHOD** implementation is being used (either set directly or as provided by an ENGINE module), then it is entirely responsible for the generation and management of a cryptographically secure PRNG stream. The mechanisms described below relate solely to the software PRNG implementation built in to OpenSSL and used by default.

These functions implement a cryptographically secure pseudo-random number generator (PRNG). It is used by other library functions for example to generate random keys, and applications can use it when they need randomness.

A cryptographic PRNG must be seeded with unpredictable data such as mouse movements or keys pressed at random by the user. This is described in *RAND_add* (3). Its state can be saved in a seed file (see *RAND_load_file* (3)) to avoid having to go through the seeding process whenever the application is started.

RAND_bytes (3) describes how to obtain random data from the PRNG.

INTERNALS

The *RAND_SSLeay()* method implements a PRNG based on a cryptographic hash function.

The following description of its design is based on the SSLeay documentation:

First up I will state the things I believe I need for a good RNG.

1 A good hashing algorithm to mix things up and to convert the RNG ’state’ to random numbers.

2 An initial source of random ’state’.

3 The state should be very large. If the RNG is being used to generate 4096 bit RSA keys, 2 2048 bit random strings are required (at a minimum). If your RNG state only has 128 bits, you are obviously limiting the search space to 128 bits, not 2048. I'm probably getting a little carried away on this last point but it does indicate that it may not be a bad idea to keep quite a lot of RNG state. It should be easier to break a cipher than guess the RNG seed data.

4 Any RNG seed data should influence all subsequent random numbers generated. This implies that any random seed data entered will have an influence on all subsequent random numbers generated.

5 When using data to seed the RNG state, the data used should not be extractable from the RNG state. I believe this should be a requirement because one possible source of 'secret' semi random data would be a private key or a password. This data must not be disclosed by either subsequent random numbers or a 'core' dump left by a program crash.

6 Given the same initial 'state', 2 systems should deviate in their RNG state (and hence the random numbers generated) over time if at all possible.

7 Given the random number output stream, it should not be possible to determine the RNG state or the next random number.

The algorithm is as follows.

There is global state made up of a 1023 byte buffer (the 'state'), a working hash value ('md'), and a counter ('count').

Whenever seed data is added, it is inserted into the 'state' as follows.

The input is chopped up into units of 20 bytes (or less for the last block). Each of these blocks is run through the hash function as follows: The data passed to the hash function is the current 'md', the same number of bytes from the 'state' (the location determined by in incremented looping index) as the current 'block', the new key data 'block', and 'count' (which is incremented after each use). The result of this is kept in 'md' and also xored into the 'state' at the same locations that were used as input into the hash function. I believe this system addresses points 1 (hash function; currently SHA−1), 3 (the 'state'), 4 (via the 'md'), 5 (by the use of a hash function and xor).

When bytes are extracted from the RNG, the following process is used. For each group of 10 bytes (or less), we do the following:

Input into the hash function the local 'md' (which is initialized from the global 'md' before any bytes are generated), the bytes that are to be overwritten by the random bytes, and bytes from the 'state' (incrementing looping index). From this digest output (which is kept in 'md'), the top (up to) 10 bytes are returned to the caller and the bottom 10 bytes are xored into the 'state'.

Finally, after we have finished 'num' random bytes for the caller, 'count' (which is incremented) and the local and global 'md' are fed into the hash function and the results are kept in the global 'md'.

I believe the above addressed points 1 (use of SHA−1), 6 (by hashing into the 'state' the 'old' data from the caller that is about to be overwritten) and 7 (by not using the 10 bytes given to the caller to update the 'state', but they are used to update 'md').

So of the points raised, only 2 is not addressed (but see *RAND_add* (3)).

SEE ALSO

BN_rand (3), *RAND_add* (3), *RAND_load_file* (3), *RAND_egd* (3), *RAND_bytes* (3), *RAND_set_rand_method* (3), *RAND_cleanup* (3)

NAME

RC4_set_key, RC4 − RC4 encryption

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rc4.h>

void RC4_set_key(RC4_KEY *key, int len, const unsigned char *data);

void RC4(RC4_KEY *key, unsigned long len, const unsigned char *indata,
         unsigned char *outdata);
```

DESCRIPTION

This library implements the Alleged RC4 cipher, which is described for example in *Applied Cryptography*. It is believed to be compatible with RC4[TM], a proprietary cipher of RSA Security Inc.

RC4 is a stream cipher with variable key length. Typically, 128 bit (16 byte) keys are used for strong encryption, but shorter insecure key sizes have been widely used due to export restrictions.

RC4 consists of a key setup phase and the actual encryption or decryption phase.

RC4_set_key() sets up the **RC4_KEY key** using the **len** bytes long key at **data**.

RC4() encrypts or decrypts the **len** bytes of data at **indata** using **key** and places the result at **outdata**. Repeated *RC4()* calls with the same **key** yield a continuous key stream.

Since RC4 is a stream cipher (the input is XORed with a pseudo-random key stream to produce the output), decryption uses the same function calls as encryption.

Applications should use the higher level functions *EVP_EncryptInit* (3) etc. instead of calling the RC4 functions directly.

RETURN VALUES

RC4_set_key() and *RC4()* do not return values.

NOTE

Certain conditions have to be observed to securely use stream ciphers. It is not permissible to perform multiple encryptions using the same key stream.

SEE ALSO

openssl_blowfish (3), *openssl_des* (3), *rc2* (3)

HISTORY

RC4_set_key() and *RC4()* are available in all versions of SSLeay and OpenSSL.

NAME

RIPEMD160, RIPEMD160_Init, RIPEMD160_Update, RIPEMD160_Final − RIPEMD−160 hash function

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ripemd.h>

unsigned char *RIPEMD160(const unsigned char *d, unsigned long n,
                 unsigned char *md);

int RIPEMD160_Init(RIPEMD160_CTX *c);
int RIPEMD160_Update(RIPEMD_CTX *c, const void *data,
                 unsigned long len);
int RIPEMD160_Final(unsigned char *md, RIPEMD160_CTX *c);
```

DESCRIPTION

RIPEMD−160 is a cryptographic hash function with a 160 bit output.

RIPEMD160() computes the RIPEMD−160 message digest of the **n** bytes at **d** and places it in **md** (which must have space for RIPEMD160_DIGEST_LENGTH == 20 bytes of output). If **md** is NULL, the digest is placed in a static array.

The following functions may be used if the message is not completely stored in memory:

RIPEMD160_Init() initializes a **RIPEMD160_CTX** structure.

RIPEMD160_Update() can be called repeatedly with chunks of the message to be hashed (**len** bytes at **data**).

RIPEMD160_Final() places the message digest in **md**, which must have space for RIPEMD160_DIGEST_LENGTH == 20 bytes of output, and erases the **RIPEMD160_CTX**.

Applications should use the higher level functions *EVP_DigestInit* (3) etc. instead of calling the hash functions directly.

RETURN VALUES

RIPEMD160() returns a pointer to the hash value.

RIPEMD160_Init(), *RIPEMD160_Update()* and *RIPEMD160_Final()* return 1 for success, 0 otherwise.

CONFORMING TO

ISO/IEC 10118−3 (draft) (??)

SEE ALSO

openssl_sha (3), *openssl_hmac* (3), *EVP_DigestInit* (3)

HISTORY

RIPEMD160(), *RIPEMD160_Init()*, *RIPEMD160_Update()* and *RIPEMD160_Final()* are available since SSLeay 0.9.0.

NAME

rsa − RSA public key cryptosystem

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/rsa.h>
#include <openssl/engine.h>

RSA * RSA_new(void);
void RSA_free(RSA *rsa);

int RSA_public_encrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa, int padding);
int RSA_private_decrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa, int padding);
int RSA_private_encrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa,int padding);
int RSA_public_decrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa,int padding);

int RSA_sign(int type, unsigned char *m, unsigned int m_len,
   unsigned char *sigret, unsigned int *siglen, RSA *rsa);
int RSA_verify(int type, unsigned char *m, unsigned int m_len,
   unsigned char *sigbuf, unsigned int siglen, RSA *rsa);

int RSA_size(const RSA *rsa);

RSA *RSA_generate_key(int num, unsigned long e,
   void (*callback)(int,int,void *), void *cb_arg);

int RSA_check_key(RSA *rsa);

int RSA_blinding_on(RSA *rsa, BN_CTX *ctx);
void RSA_blinding_off(RSA *rsa);

void RSA_set_default_method(const RSA_METHOD *meth);
const RSA_METHOD *RSA_get_default_method(void);
int RSA_set_method(RSA *rsa, const RSA_METHOD *meth);
const RSA_METHOD *RSA_get_method(const RSA *rsa);
RSA_METHOD *RSA_PKCS1_SSLeay(void);
RSA_METHOD *RSA_null_method(void);
int RSA_flags(const RSA *rsa);
RSA *RSA_new_method(ENGINE *engine);

int RSA_print(BIO *bp, RSA *x, int offset);
int RSA_print_fp(FILE *fp, RSA *x, int offset);

int RSA_get_ex_new_index(long argl, char *argp, int (*new_func)(),
   int (*dup_func)(), void (*free_func)());
int RSA_set_ex_data(RSA *r,int idx,char *arg);
char *RSA_get_ex_data(RSA *r, int idx);

int RSA_sign_ASN1_OCTET_STRING(int dummy, unsigned char *m,
   unsigned int m_len, unsigned char *sigret, unsigned int *siglen,
   RSA *rsa);
int RSA_verify_ASN1_OCTET_STRING(int dummy, unsigned char *m,
   unsigned int m_len, unsigned char *sigbuf, unsigned int siglen,
   RSA *rsa);
```

DESCRIPTION

These functions implement RSA public key encryption and signatures as defined in PKCS #1 v2.0 [RFC 2437].

The **RSA** structure consists of several BIGNUM components. It can contain public as well as private RSA keys:

```
struct
       {
       BIGNUM *n;              // public modulus
       BIGNUM *e;              // public exponent
       BIGNUM *d;              // private exponent
       BIGNUM *p;              // secret prime factor
       BIGNUM *q;              // secret prime factor
       BIGNUM *dmp1;           // d mod (p-1)
       BIGNUM *dmq1;           // d mod (q-1)
       BIGNUM *iqmp;           // q^-1 mod p
       // ...
       };
RSA
```

In public keys, the private exponent and the related secret values are **NULL**.

p, **q**, **dmp1**, **dmq1** and **iqmp** may be **NULL** in private keys, but the RSA operations are much faster when these values are available.

Note that RSA keys may use non-standard **RSA_METHOD** implementations, either directly or by the use of **ENGINE** modules. In some cases (eg. an ENGINE providing support for hardware-embedded keys), these BIGNUM values will not be used by the implementation or may be used for alternative data storage. For this reason, applications should generally avoid using RSA structure elements directly and instead use API functions to query or modify keys.

CONFORMING TO

SSL, PKCS #1 v2.0

PATENTS

RSA was covered by a US patent which expired in September 2000.

SEE ALSO

openssl_rsa (1), *openssl_bn* (3), *openssl_dsa* (3), *openssl_dh* (3), *openssl_rand* (3), *engine* (3), *RSA_new* (3), *RSA_public_encrypt* (3), *RSA_sign* (3), *RSA_size* (3), *RSA_generate_key* (3), *RSA_check_key* (3), *RSA_blinding_on* (3), *RSA_set_method* (3), *RSA_print* (3), *RSA_get_ex_new_index* (3), *RSA_private_encrypt* (3), *RSA_sign_ASN1_OCTET_STRING* (3), *RSA_padding_add_PKCS1_type_1* (3)

NAME

SHA1, SHA1_Init, SHA1_Update, SHA1_Final − Secure Hash Algorithm

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/sha.h>

unsigned char *SHA1(const unsigned char *d, unsigned long n,
                    unsigned char *md);

int SHA1_Init(SHA_CTX *c);
int SHA1_Update(SHA_CTX *c, const void *data,
                    unsigned long len);
int SHA1_Final(unsigned char *md, SHA_CTX *c);
```

DESCRIPTION

SHA−1 (Secure Hash Algorithm) is a cryptographic hash function with a 160 bit output.

SHA1() computes the SHA−1 message digest of the **n** bytes at **d** and places it in **md** (which must have space for SHA_DIGEST_LENGTH == 20 bytes of output). If **md** is NULL, the digest is placed in a static array.

The following functions may be used if the message is not completely stored in memory:

SHA1_Init() initializes a **SHA_CTX** structure.

SHA1_Update() can be called repeatedly with chunks of the message to be hashed (**len** bytes at **data**).

SHA1_Final() places the message digest in **md**, which must have space for SHA_DIGEST_LENGTH == 20 bytes of output, and erases the **SHA_CTX**.

Applications should use the higher level functions *EVP_DigestInit* (3) etc. instead of calling the hash functions directly.

The predecessor of SHA−1, SHA, is also implemented, but it should be used only when backward compatibility is required.

RETURN VALUES

SHA1() returns a pointer to the hash value.

SHA1_Init(), *SHA1_Update()* and *SHA1_Final()* return 1 for success, 0 otherwise.

CONFORMING TO

SHA: US Federal Information Processing Standard FIPS PUB 180 (Secure Hash Standard), SHA−1: US Federal Information Processing Standard FIPS PUB 180−1 (Secure Hash Standard), ANSI X9.30

SEE ALSO

openssl_ripemd (3), *openssl_hmac* (3), *EVP_DigestInit* (3)

HISTORY

SHA1(), *SHA1_Init()*, *SHA1_Update()* and *SHA1_Final()* are available in all versions of SSLeay and OpenSSL.

NAME

CRYPTO_set_locking_callback, CRYPTO_set_id_callback, CRYPTO_set_idptr_callback, CRYPTO_num_locks, CRYPTO_set_dynlock_create_callback, CRYPTO_set_dynlock_lock_callback, CRYPTO_set_dynlock_destroy_callback, CRYPTO_get_new_dynlockid, CRYPTO_destroy_dynlockid, CRYPTO_lock − OpenSSL thread support

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/crypto.h>

void CRYPTO_set_locking_callback(void (*locking_function)(int mode,
       int n, const char *file, int line));

void CRYPTO_set_id_callback(unsigned long (*id_function)(void));

void CRYPTO_set_idptr_callback(void *(*idptr_function)(void));

int CRYPTO_num_locks(void);

/* struct CRYPTO_dynlock_value needs to be defined by the user */
struct CRYPTO_dynlock_value;

void CRYPTO_set_dynlock_create_callback(struct CRYPTO_dynlock_value *
       (*dyn_create_function)(char *file, int line));
void CRYPTO_set_dynlock_lock_callback(void (*dyn_lock_function)
       (int mode, struct CRYPTO_dynlock_value *l,
       const char *file, int line));
void CRYPTO_set_dynlock_destroy_callback(void (*dyn_destroy_function)
       (struct CRYPTO_dynlock_value *l, const char *file, int line));

int CRYPTO_get_new_dynlockid(void);

void CRYPTO_destroy_dynlockid(int i);

void CRYPTO_lock(int mode, int n, const char *file, int line);

#define CRYPTO_w_lock(type)    \
       CRYPTO_lock(CRYPTO_LOCK|CRYPTO_WRITE,type,__FILE__,__LINE__)
#define CRYPTO_w_unlock(type)  \
       CRYPTO_lock(CRYPTO_UNLOCK|CRYPTO_WRITE,type,__FILE__,__LINE__)
#define CRYPTO_r_lock(type)    \
       CRYPTO_lock(CRYPTO_LOCK|CRYPTO_READ,type,__FILE__,__LINE__)
#define CRYPTO_r_unlock(type)  \
       CRYPTO_lock(CRYPTO_UNLOCK|CRYPTO_READ,type,__FILE__,__LINE__)
#define CRYPTO_add(addr,amount,type)   \
       CRYPTO_add_lock(addr,amount,type,__FILE__,__LINE__)
```

DESCRIPTION

OpenSSL can safely be used in multi-threaded applications provided that at least two callback functions are set.

locking_function(int mode, int n, const char *file, int line) is needed to perform locking on shared data structures. (Note that OpenSSL uses a number of global data structures that will be implicitly shared whenever multiple threads use OpenSSL.) Multi-threaded applications will crash at random if it is not set.

locking_function() must be able to handle up to *CRYPTO_num_locks()* different mutex locks. It sets the **n**−th lock if **mode** & **CRYPTO_LOCK**, and releases it otherwise.

file and **line** are the file number of the function setting the lock. They can be useful for debugging.

id_function(void) is a function that returns a numerical thread ID, for example *pthread_self()* if it returns an integer (see NOTES below). By OpenSSL's defaults, this is not needed on Windows nor on platforms where *getpid()* returns a different ID for each thread (see NOTES below).

idptr_function(void) is a function that similarly returns a thread ID, but of type void *. This is not needed on platforms where &errno is different for each thread.

Additionally, OpenSSL supports dynamic locks, and sometimes, some parts of OpenSSL need it for better performance. To enable this, the following is required:

* Three additional callback function, dyn_create_function, dyn_lock_function and dyn_destroy_function.
* A structure defined with the data that each lock needs to handle.

struct CRYPTO_dynlock_value has to be defined to contain whatever structure is needed to handle locks.

dyn_create_function(const char *file, int line) is needed to create a lock. Multi-threaded applications might crash at random if it is not set.

dyn_lock_function(int mode, CRYPTO_dynlock *l, const char *file, int line) is needed to perform locking off dynamic lock numbered n. Multi-threaded applications might crash at random if it is not set.

dyn_destroy_function(CRYPTO_dynlock *l, const char *file, int line) is needed to destroy the lock l. Multi-threaded applications might crash at random if it is not set.

CRYPTO_get_new_dynlockid() is used to create locks. It will call dyn_create_function for the actual creation.

CRYPTO_destroy_dynlockid() is used to destroy locks. It will call dyn_destroy_function for the actual destruction.

CRYPTO_lock() is used to lock and unlock the locks. mode is a bitfield describing what should be done with the lock. n is the number of the lock as returned from *CRYPTO_get_new_dynlockid()*. mode can be combined from the following values. These values are pairwise exclusive, with undefined behaviour if misused (for example, CRYPTO_READ and CRYPTO_WRITE should not be used together):

```
CRYPTO_LOCK     0x01
CRYPTO_UNLOCK   0x02
CRYPTO_READ     0x04
CRYPTO_WRITE    0x08
```

RETURN VALUES

CRYPTO_num_locks() returns the required number of locks.

CRYPTO_get_new_dynlockid() returns the index to the newly created lock.

The other functions return no values.

NOTES

You can find out if OpenSSL was configured with thread support:

```
#define OPENSSL_THREAD_DEFINES
#include <openssl/opensslconf.h>
#if defined(OPENSSL_THREADS)
  // thread support enabled
#else
  // no thread support
#endif
```

Also, dynamic locks are currently not used internally by OpenSSL, but may do so in the future.

Defining id_function(void) has it's own issues. Generally speaking, *pthread_self()* should be used, even on platforms where *getpid()* gives different answers in each thread, since that may depend on the machine the program is run on, not the machine where the program is being compiled. For instance, Red Hat 8 Linux and earlier used LinuxThreads, whose *getpid()* returns a different value for each thread. Red Hat 9 Linux and later use NPTL, which is Posix−conformant, and has a *getpid()* that returns the same value for all threads in a process. A program compiled on Red Hat 8 and run on Red Hat 9 will therefore see *getpid()* returning the same value for all threads.

There is still the issue of platforms where *pthread_self()* returns something other than an integer. It is for

cases like this that *CRYPTO_set_idptr_callback()* comes in handy. (E.g., call *malloc*(1) once in each thread, and have *idptr_function()* return a pointer to this object.) Note that if neither *id_function()* or *idptr_function()* are provided, OpenSSL will use (&errno) as a fallback (as this usually returns a unique address for each thread).

EXAMPLES

crypto/threads/mttest.c shows examples of the callback functions on Solaris, Irix and Win32.

HISTORY

CRYPTO_set_locking_callback() and *CRYPTO_set_id_callback()* are available in all versions of SSLeay and OpenSSL. *CRYPTO_num_locks()* was added in OpenSSL 0.9.4. All functions dealing with dynamic locks were added in OpenSSL 0.9.5b−dev.

CRYPTO_set_idptr_callback() was added in OpenSSL 0.9.9.

SEE ALSO

crypto(3)

NAME

UI_new, UI_new_method, UI_free, UI_add_input_string, UI_dup_input_string, UI_add_verify_string, UI_dup_verify_string, UI_add_input_boolean, UI_dup_input_boolean, UI_add_info_string, UI_dup_info_string, UI_add_error_string, UI_dup_error_string, UI_construct_prompt, UI_add_user_data, UI_get0_user_data, UI_get0_result, UI_process, UI_ctrl, UI_set_default_method, UI_get_default_method, UI_get_method, UI_set_method, UI_OpenSSL, ERR_load_UI_strings − New User Interface

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/ui.h>

typedef struct ui_st UI;
typedef struct ui_method_st UI_METHOD;

UI *UI_new(void);
UI *UI_new_method(const UI_METHOD *method);
void UI_free(UI *ui);

int UI_add_input_string(UI *ui, const char *prompt, int flags,
       char *result_buf, int minsize, int maxsize);
int UI_dup_input_string(UI *ui, const char *prompt, int flags,
       char *result_buf, int minsize, int maxsize);
int UI_add_verify_string(UI *ui, const char *prompt, int flags,
       char *result_buf, int minsize, int maxsize, const char *test_buf);
int UI_dup_verify_string(UI *ui, const char *prompt, int flags,
       char *result_buf, int minsize, int maxsize, const char *test_buf);
int UI_add_input_boolean(UI *ui, const char *prompt, const char *action_desc,
       const char *ok_chars, const char *cancel_chars,
       int flags, char *result_buf);
int UI_dup_input_boolean(UI *ui, const char *prompt, const char *action_desc,
       const char *ok_chars, const char *cancel_chars,
       int flags, char *result_buf);
int UI_add_info_string(UI *ui, const char *text);
int UI_dup_info_string(UI *ui, const char *text);
int UI_add_error_string(UI *ui, const char *text);
int UI_dup_error_string(UI *ui, const char *text);

/* These are the possible flags.  They can be or'ed together. */
#define UI_INPUT_FLAG_ECHO              0x01
#define UI_INPUT_FLAG_DEFAULT_PWD       0x02

char *UI_construct_prompt(UI *ui_method,
       const char *object_desc, const char *object_name);

void *UI_add_user_data(UI *ui, void *user_data);
void *UI_get0_user_data(UI *ui);

const char *UI_get0_result(UI *ui, int i);

int UI_process(UI *ui);

int UI_ctrl(UI *ui, int cmd, long i, void *p, void (*f)());
#define UI_CTRL_PRINT_ERRORS            1
#define UI_CTRL_IS_REDOABLE             2

void UI_set_default_method(const UI_METHOD *meth);
const UI_METHOD *UI_get_default_method(void);
const UI_METHOD *UI_get_method(UI *ui);
const UI_METHOD *UI_set_method(UI *ui, const UI_METHOD *meth);
```

```
UI_METHOD *UI_OpenSSL(void);
```

DESCRIPTION

UI stands for User Interface, and is general purpose set of routines to prompt the user for text-based information. Through user-written methods (see *ui_create*(3)), prompting can be done in any way imaginable, be it plain text prompting, through dialog boxes or from a cell phone.

All the functions work through a context of the type UI. This context contains all the information needed to prompt correctly as well as a reference to a UI_METHOD, which is an ordered vector of functions that carry out the actual prompting.

The first thing to do is to create a UI with *UI_new()* or *UI_new_method()*, then add information to it with the UI_add or UI_dup functions. Also, user-defined random data can be passed down to the underlying method through calls to UI_add_user_data. The default UI method doesn't care about these data, but other methods might. Finally, use *UI_process()* to actually perform the prompting and *UI_get0_result()* to find the result to the prompt.

A UI can contain more than one prompt, which are performed in the given sequence. Each prompt gets an index number which is returned by the UI_add and UI_dup functions, and has to be used to get the corresponding result with *UI_get0_result()*.

The functions are as follows:

UI_new() creates a new UI using the default UI method. When done with this UI, it should be freed using *UI_free()*.

UI_new_method() creates a new UI using the given UI method. When done with this UI, it should be freed using *UI_free()*.

UI_OpenSSL() returns the built-in UI method (note: not the default one, since the default can be changed. See further on). This method is the most machine/OS dependent part of OpenSSL and normally generates the most problems when porting.

UI_free() removes a UI from memory, along with all other pieces of memory that's connected to it, like duplicated input strings, results and others.

UI_add_input_string() and *UI_add_verify_string()* add a prompt to the UI, as well as flags and a result buffer and the desired minimum and maximum sizes of the result. The given information is used to prompt for information, for example a password, and to verify a password (i.e. having the user enter it twice and check that the same string was entered twice). *UI_add_verify_string()* takes and extra argument that should be a pointer to the result buffer of the input string that it's supposed to verify, or verification will fail.

UI_add_input_boolean() adds a prompt to the UI that's supposed to be answered in a boolean way, with a single character for yes and a different character for no. A set of characters that can be used to cancel the prompt is given as well. The prompt itself is really divided in two, one part being the descriptive text (given through the *prompt* argument) and one describing the possible answers (given through the *action_desc* argument).

UI_add_info_string() and *UI_add_error_string()* add strings that are shown at the same time as the prompt for extra information or to show an error string. The difference between the two is only conceptual. With the builtin method, there's no technical difference between them. Other methods may make a difference between them, however.

The flags currently supported are UI_INPUT_FLAG_ECHO, which is relevant for *UI_add_input_string()* and will have the users response be echoed (when prompting for a password, this flag should obviously not be used, and UI_INPUT_FLAG_DEFAULT_PWD, which means that a default password of some sort will be used (completely depending on the application and the UI method).

UI_dup_input_string(), *UI_dup_verify_string()*, *UI_dup_input_boolean()*, *UI_dup_info_string()* and *UI_dup_error_string()* are basically the same as their UI_add counterparts, except that they make their own copies of all strings.

UI_construct_prompt() is a helper function that can be used to create a prompt from two pieces of

information: an description and a name. The default constructor (if there is none provided by the method used) creates a string "Enter *description* for *name*:‘‘. With the description ’’pass phrase‘‘ and the file name ’’foo.key‘‘, that becomes ’’Enter pass phrase for foo.key:". Other methods may create whatever string and may include encodings that will be processed by the other method functions.

UI_add_user_data() adds a piece of memory for the method to use at any time. The builtin UI method doesn’t care about this info. Note that several calls to this function doesn’t add data, it replaces the previous blob with the one given as argument.

UI_get0_user_data() retrieves the data that has last been given to the UI with *UI_add_user_data()*.

UI_get0_result() returns a pointer to the result buffer associated with the information indexed by *i*.

UI_process() goes through the information given so far, does all the printing and prompting and returns.

UI_ctrl() adds extra control for the application author. For now, it understands two commands: UI_CTRL_PRINT_ERRORS, which makes *UI_process()* print the OpenSSL error stack as part of processing the UI, and UI_CTRL_IS_REDOABLE, which returns a flag saying if the used UI can be used again or not.

UI_set_default_method() changes the default UI method to the one given.

UI_get_default_method() returns a pointer to the current default UI method.

UI_get_method() returns the UI method associated with a given UI.

UI_set_method() changes the UI method associated with a given UI.

SEE ALSO

ui_create (3), *ui_compat* (3)

HISTORY

The UI section was first introduced in OpenSSL 0.9.7.

AUTHOR

Richard Levitte (richard@levitte.org) for the OpenSSL project (http://www.openssl.org).

NAME

des_read_password, des_read_2passwords, des_read_pw_string, des_read_pw − Compatibility user interface functions

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/des_old.h>

int des_read_password(DES_cblock *key,const char *prompt,int verify);
int des_read_2passwords(DES_cblock *key1,DES_cblock *key2,
       const char *prompt,int verify);

int des_read_pw_string(char *buf,int length,const char *prompt,int verify);
int des_read_pw(char *buf,char *buff,int size,const char *prompt,int verify);
```

DESCRIPTION

The DES library contained a few routines to prompt for passwords. These aren't necessarely dependent on DES, and have therefore become part of the UI compatibility library.

des_read_pw() writes the string specified by *prompt* to standard output turns echo off and reads an input string from the terminal. The string is returned in *buf*, which must have spac for at least *size* bytes. If *verify* is set, the user is asked for the password twice and unless the two copies match, an error is returned. The second password is stored in *buff*, which must therefore also be at least *size* bytes. A return code of −1 indicates a system error, 1 failure due to use interaction, and 0 is success. All other functions described here use *des_read_pw()* to do the work.

des_read_pw_string() is a variant of *des_read_pw()* that provides a buffer for you if *verify* is set.

des_read_password() calls *des_read_pw()* and converts the password to a DES key by calling *DES_string_to_key()*; *des_read_2password()* operates in the same way as *des_read_password()* except that it generates two keys by using the *DES_string_to_2key()* function.

NOTES

des_read_pw_string() is available in the MIT Kerberos library as well, and is also available under the name *EVP_read_pw_string()*.

SEE ALSO

ui (3), *ui_create* (3)

AUTHOR

Richard Levitte (richard@levitte.org) for the OpenSSL project (http://www.openssl.org).

NAME

x509 − X.509 certificate handling

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

```
#include <openssl/x509.h>
```

DESCRIPTION

A X.509 certificate is a structured grouping of information about an individual, a device, or anything one can imagine. A X.509 CRL (certificate revocation list) is a tool to help determine if a certificate is still valid. The exact definition of those can be found in the X.509 document from ITU−T, or in RFC3280 from PKIX. In OpenSSL, the type X509 is used to express such a certificate, and the type X509_CRL is used to express a CRL.

A related structure is a certificate request, defined in PKCS#10 from RSA Security, Inc, also reflected in RFC2896. In OpenSSL, the type X509_REQ is used to express such a certificate request.

To handle some complex parts of a certificate, there are the types X509_NAME (to express a certificate name), X509_ATTRIBUTE (to express a certificate attributes), X509_EXTENSION (to express a certificate extension) and a few more.

Finally, there's the supertype X509_INFO, which can contain a CRL, a certificate and a corresponding private key.

X509_..., **d2i_X509_...** and **i2d_X509_...** handle X.509 certificates, with some exceptions, shown below.

X509_CRL_..., **d2i_X509_CRL_...** and **i2d_X509_CRL_...** handle X.509 CRLs.

X509_REQ_..., **d2i_X509_REQ_...** and **i2d_X509_REQ_...** handle PKCS#10 certificate requests.

X509_NAME_... handle certificate names.

X509_ATTRIBUTE_... handle certificate attributes.

X509_EXTENSION_... handle certificate extensions.

SEE ALSO

X509_NAME_ENTRY_get_object (3), *X509_NAME_add_entry_by_txt* (3), *X509_NAME_add_entry_by_NID* (3), *X509_NAME_print_ex* (3), *X509_NAME_new* (3), *d2i_X509* (3), *d2i_X509_ALGOR* (3), *d2i_X509_CRL* (3), *d2i_X509_NAME* (3), *d2i_X509_REQ* (3), *d2i_X509_SIG* (3), *crypto* (3), *x509v3* (3)

NAME

ossaudio — OSS audio emulation

LIBRARY

OSS Audio Emulation Library (libossaudio, −lossaudio)

SYNOPSIS

```
#include <soundcard.h>
```

DESCRIPTION

The **ossaudio** library provides an emulation of the OSS (Linux) audio interface.

Use the native interface for new programs and the emulation library only for porting programs.

SEE ALSO

audio(4), midi(4)

HISTORY

The **ossaudio** library first appeared in NetBSD 1.3.

BUGS

The emulation uses a #define for **ioctl**() so some obscure programs can fail to compile.

The emulation is incomplete.

The emulation only covers **ioctl**(), there are other differences as well. E.g., on a write that would block in non-blocking mode Linux returns EINTR whereas NetBSD 1.3 returns EAGAIN.

NAME

pam_acct_mgmt, **pam_authenticate**, **pam_chauthtok**, **pam_close_session**, **pam_end**, **pam_get_data**, **pam_get_item**, **pam_get_user**, **pam_getenv**, **pam_getenvlist**, **pam_open_session**, **pam_putenv**, **pam_set_data**, **pam_set_item**, **pam_setcred**, **pam_start**, **pam_strerror** — Pluggable Authentication Modules Library

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <security/pam_appl.h>

int
pam_acct_mgmt(pam_handle_t *pamh, int flags);

int
pam_authenticate(pam_handle_t *pamh, int flags);

int
pam_chauthtok(pam_handle_t *pamh, int flags);

int
pam_close_session(pam_handle_t *pamh, int flags);

int
pam_end(pam_handle_t *pamh, int status);

int
pam_get_data(const pam_handle_t *pamh, const char *module_data_name,
    const void **data);

int
pam_get_item(const pam_handle_t *pamh, int item_type, const void **item);

int
pam_get_user(pam_handle_t *pamh, const char **user, const char *prompt);

const char *
pam_getenv(pam_handle_t *pamh, const char *name);

char **
pam_getenvlist(pam_handle_t *pamh);

int
pam_open_session(pam_handle_t *pamh, int flags);

int
pam_putenv(pam_handle_t *pamh, const char *namevalue);

int
pam_set_data(pam_handle_t *pamh, const char *module_data_name, void *data,
    void (*cleanup)(pam_handle_t *pamh, void *data, int pam_end_status));

int
pam_set_item(pam_handle_t *pamh, int item_type, const void *item);

int
pam_setcred(pam_handle_t *pamh, int flags);
```

```
int
pam_start(const char *service, const char *user,
    const struct pam_conv *pam_conv, pam_handle_t **pamh);

const char *
pam_strerror(const pam_handle_t *pamh, int error_number);
```

DESCRIPTION

The Pluggable Authentication Modules (PAM) library abstracts a number of common authentication-related operations and provides a framework for dynamically loaded modules that implement these operations in various ways.

Terminology

In PAM parlance, the application that uses PAM to authenticate a user is the server, and is identified for configuration purposes by a service name, which is often (but not necessarily) the program name.

The user requesting authentication is called the applicant, while the user (usually, root) charged with verifying his identity and granting him the requested credentials is called the arbitrator.

The sequence of operations the server goes through to authenticate a user and perform whatever task he requested is a PAM transaction; the context within which the server performs the requested task is called a session.

The functionality embodied by PAM is divided into six primitives grouped into four facilities: authentication, account management, session management and password management.

Conversation

The PAM library expects the application to provide a conversation callback which it can use to communicate with the user. Some modules may use specialized conversation functions to communicate with special hardware such as cryptographic dongles or biometric devices. See pam_conv(3) for details.

Initialization and Cleanup

The **pam_start**() function initializes the PAM library and returns a handle which must be provided in all subsequent function calls. The transaction state is contained entirely within the structure identified by this handle, so it is possible to conduct multiple transactions in parallel.

The **pam_end**() function releases all resources associated with the specified context, and can be called at any time to terminate a PAM transaction.

Storage

The **pam_set_item**() and **pam_get_item**() functions set and retrieve a number of predefined items, including the service name, the names of the requesting and target users, the conversation function, and prompts.

The **pam_set_data**() and **pam_get_data**() functions manage named chunks of free-form data, generally used by modules to store state from one invocation to another.

Authentication

There are two authentication primitives: **pam_authenticate**() and **pam_setcred**(). The former authenticates the user, while the latter manages his credentials.

Account Management

The **pam_acct_mgmt**() function enforces policies such as password expiry, account expiry, time-of-day restrictions, and so forth.

Session Management
The **pam_open_session**() and **pam_close_session**() functions handle session setup and teardown.

Password Management
The **pam_chauthtok**() function allows the server to change the user's password, either at the user's request or because the password has expired.

Miscellaneous
The **pam_putenv**(), **pam_getenv**() and **pam_getenvlist**() functions manage a private environment list in which modules can set environment variables they want the server to export during the session.

The **pam_strerror**() function returns a pointer to a string describing the specified PAM error code.

RETURN VALUES

The following return codes are defined by <security/pam_constants.h>:

[PAM_ABORT] General failure.

[PAM_ACCT_EXPIRED]
User account has expired.

[PAM_AUTHINFO_UNAVAIL]
Authentication information is unavailable.

[PAM_AUTHTOK_DISABLE_AGING]
Authentication token aging disabled.

[PAM_AUTHTOK_ERR]
Authentication token failure.

[PAM_AUTHTOK_EXPIRED]
Password has expired.

[PAM_AUTHTOK_LOCK_BUSY]
Authentication token lock busy.

[PAM_AUTHTOK_RECOVERY_ERR]
Failed to recover old authentication token.

[PAM_AUTH_ERR] Authentication error.

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_CRED_ERR] Failed to set user credentials.

[PAM_CRED_EXPIRED]
User credentials have expired.

[PAM_CRED_INSUFFICIENT]
Insufficient credentials.

[PAM_CRED_UNAVAIL]
Failed to retrieve user credentials.

[PAM_DOMAIN_UNKNOWN]
Unknown authentication domain.

[PAM_IGNORE] Ignore this module.

[PAM_MAXTRIES] Maximum number of tries exceeded.

[PAM_MODULE_UNKNOWN]
Unknown module type.

[PAM_NEW_AUTHTOK_REQD]
New authentication token required.

[PAM_NO_MODULE_DATA]
Module data not found.

[PAM_OPEN_ERR] Failed to load module.

[PAM_PERM_DENIED]
Permission denied.

[PAM_SERVICE_ERR]
Error in service module.

[PAM_SESSION_ERR]
Session failure.

[PAM_SUCCESS] Success.

[PAM_SYMBOL_ERR]
Invalid symbol.

[PAM_SYSTEM_ERR]
System error.

[PAM_TRY_AGAIN] Try again.

[PAM_USER_UNKNOWN]
Unknown user.

SEE ALSO

openpam(3), pam_acct_mgmt(3), pam_authenticate(3), pam_chauthtok(3), pam_close_session(3), pam_conv(3), pam_end(3), pam_get_data(3), pam_getenv(3), pam_getenvlist(3), pam_get_item(3), pam_get_user(3), pam_open_session(3), pam_putenv(3), pam_setcred(3), pam_set_data(3), pam_set_item(3), pam_start(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The OpenPAM library and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_acct_mgmt — perform PAM account validation procedures

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_acct_mgmt(pam_handle_t *pamh, int flags);
```

DESCRIPTION

The **pam_acct_mgmt** function verifies and enforces account restrictions after the user has been authenticated.

The *flags* argument is the binary or of zero or more of the following values:

PAM_SILENT
: Do not emit any messages.

PAM_DISALLOW_NULL_AUTHTOK
: Fail if the user's authentication token is null.

If any other bits are set, **pam_acct_mgmt** will return PAM_SYMBOL_ERR.

RETURN VALUES

The **pam_acct_mgmt** function returns one of the following values:

[PAM_ABORT]
: General failure.

[PAM_ACCT_EXPIRED]
: User account has expired.

[PAM_AUTH_ERR]
: Authentication error.

[PAM_BUF_ERR]
: Memory buffer error.

[PAM_CONV_ERR]
: Conversation failure.

[PAM_NEW_AUTHTOK_REQD]
: New authentication token required.

[PAM_PERM_DENIED]
: Permission denied.

[PAM_SERVICE_ERR]
: Error in service module.

[PAM_SYSTEM_ERR]
: System error.

[PAM_USER_UNKNOWN]
: Unknown user.

SEE ALSO

pam(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_acct_mgmt** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_authenticate — perform authentication within the PAM framework

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_authenticate(pam_handle_t *pamh, int flags);
```

DESCRIPTION

The **pam_authenticate** function attempts to authenticate the user associated with the pam context specified by the *pamh* argument.

The application is free to call **pam_authenticate** as many times as it wishes, but some modules may maintain an internal retry counter and return PAM_MAXTRIES when it exceeds some preset or hardcoded limit.

The *flags* argument is the binary or of zero or more of the following values:

PAM_SILENT — Do not emit any messages.

PAM_DISALLOW_NULL_AUTHTOK — Fail if the user's authentication token is null.

If any other bits are set, **pam_authenticate** will return PAM_SYMBOL_ERR.

RETURN VALUES

The **pam_authenticate** function returns one of the following values:

[PAM_ABORT] — General failure.

[PAM_AUTHINFO_UNAVAIL] — Authentication information is unavailable.

[PAM_AUTH_ERR] — Authentication error.

[PAM_BUF_ERR] — Memory buffer error.

[PAM_CONV_ERR] — Conversation failure.

[PAM_CRED_INSUFFICIENT] — Insufficient credentials.

[PAM_MAXTRIES] — Maximum number of tries exceeded.

[PAM_PERM_DENIED] — Permission denied.

[PAM_SERVICE_ERR] — Error in service module.

[PAM_SYMBOL_ERR] — Invalid symbol.

[PAM_SYSTEM_ERR]
System error.

[PAM_USER_UNKNOWN]
Unknown user.

SEE ALSO

pam(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_authenticate** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_chauthtok — perform password related functions within the PAM framework

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_chauthtok(pam_handle_t *pamh, int flags);
```

DESCRIPTION

The **pam_chauthtok** function attempts to change the authentication token for the user associated with the pam context specified by the *pamh* argument.

The *flags* argument is the binary or of zero or more of the following values:

PAM_SILENT — Do not emit any messages.

PAM_CHANGE_EXPIRED_AUTHTOK — Change only those authentication tokens that have expired.

If any other bits are set, **pam_chauthtok** will return PAM_SYMBOL_ERR.

RETURN VALUES

The **pam_chauthtok** function returns one of the following values:

[PAM_ABORT] — General failure.

[PAM_AUTHTOK_DISABLE_AGING] — Authentication token aging disabled.

[PAM_AUTHTOK_ERR] — Authentication token failure.

[PAM_AUTHTOK_LOCK_BUSY] — Authentication token lock busy.

[PAM_AUTHTOK_RECOVERY_ERR] — Failed to recover old authentication token.

[PAM_BUF_ERR] — Memory buffer error.

[PAM_CONV_ERR] — Conversation failure.

[PAM_PERM_DENIED] — Permission denied.

[PAM_SERVICE_ERR] — Error in service module.

[PAM_SYMBOL_ERR] — Invalid symbol.

[PAM_SYSTEM_ERR] — System error.

[PAM_TRY_AGAIN] Try again.

SEE ALSO

pam(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_chauthtok** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_close_session — close an existing user session

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_close_session(pam_handle_t *pamh, int flags);
```

DESCRIPTION

The **pam_close_session** function tears down the user session previously set up by pam_open_session(3).

The *flags* argument is the binary or of zero or more of the following values:

PAM_SILENT Do not emit any messages.

If any other bits are set, **pam_close_session** will return PAM_SYMBOL_ERR.

RETURN VALUES

The **pam_close_session** function returns one of the following values:

[PAM_ABORT] General failure.

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_PERM_DENIED] Permission denied.

[PAM_SERVICE_ERR] Error in service module.

[PAM_SESSION_ERR] Session failure.

[PAM_SYMBOL_ERR] Invalid symbol.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_open_session(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_close_session** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_conv — PAM conversation system

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <security/pam_appl.h>

struct pam_message {
        int      msg_style;
        char    *msg;
};

struct pam_response {
        char    *resp;
        int      resp_retcode;
};

struct pam_conv {
        int     (*conv)(int, const struct pam_message **,
            struct pam_response **, void *);
        void    *appdata_ptr;
};
```

DESCRIPTION

The PAM library uses an application-defined callback to communicate with the user. This callback is specified by the *struct pam_conv* passed to **pam_start**() at the start of the transaction. It is also possible to set or change the conversation function at any point during a PAM transaction by changing the value of the PAM_CONV item.

The conversation function's first argument specifies the number of messages (up to PAM_NUM_MSG) to process. The second argument is a pointer to an array of pointers to *pam_message* structures containing the actual messages.

Each message can have one of four types, specified by the *msg_style* member of *struct pam_message*:

PAM_PROMPT_ECHO_OFF
: Display a prompt and accept the user's response without echoing it to the terminal. This is commonly used for passwords.

PAM_PROMPT_ECHO_ON
: Display a prompt and accept the user's response, echoing it to the terminal. This is commonly used for login names and one-time passphrases.

PAM_ERROR_MSG
: Display an error message.

PAM_TEXT_INFO
: Display an informational message.

In each case, the prompt or message to display is pointed to by the *msg* member of *struct pam_message*. It can be up to PAM_MAX_MSG_SIZE characters long, including the terminating NUL.

On success, the conversation function should allocate and fill a contiguous array of *struct pam_response*, one for each message that was passed in. A pointer to the user's response to each message (or NULL in the case of informational or error messages) should be stored in the *resp* member of the corresponding *struct pam_response*. Each response can be up to PAM_MAX_RESP_SIZE characters

long, including the terminating NUL.

The *resp_retcode* member of `struct pam_response` is unused and should be set to zero.

The conversation function should store a pointer to this array in the location pointed to by its third argument. It is the caller's responsibility to release both this array and the responses themselves, using `free`(3). It is the conversation function's responsibility to ensure that it is legal to do so.

The *appdata_ptr* member of `struct pam_conv` is passed unmodified to the conversation function as its fourth and final argument.

On failure, the conversation function should release any resources it has allocated, and return one of the predefined PAM error codes.

RETURN VALUES

The conversation function should return one of the following values:

`[PAM_BUF_ERR]` Memory buffer error.

`[PAM_CONV_ERR]` Conversation failure.

`[PAM_SUCCESS]` Success.

`[PAM_SYSTEM_ERR]` System error.

SEE ALSO

`openpam_nullconv`(3), `openpam_ttyconv`(3), `pam`(3), `pam_error`(3), `pam_get_item`(3), `pam_info`(3), `pam_prompt`(3), `pam_set_item`(3), `pam_start`(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The OpenPAM library and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_end — terminate the PAM transaction

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_end(pam_handle_t *pamh, int status);
```

DESCRIPTION

The **pam_end** function terminates a PAM transaction and destroys the corresponding PAM context, releasing all resources allocated to it.

The *status* argument should be set to the error code returned by the last API call before the call to **pam_end**

RETURN VALUES

The **pam_end** function returns one of the following values:

[PAM_SYSTEM_ERR]
: System error.

SEE ALSO

pam(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_end** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_error — display an error message

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_error(const pam_handle_t *pamh, const char *fmt, ...);
```

DESCRIPTION

The **pam_error** function displays an error message through the intermediary of the given PAM context's conversation function.

RETURN VALUES

The **pam_error** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_info(3), pam_prompt(3), pam_strerror(3), pam_verror(3)

STANDARDS

The **pam_error** function is an OpenPAM extension.

AUTHORS

The **pam_error** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_get_authtok — retrieve authentication token

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_get_authtok(pam_handle_t *pamh, int item, const char **authtok,
    const char *prompt);
```

DESCRIPTION

The **pam_get_authtok** function returns the cached authentication token, or prompts the user if no token is currently cached. Either way, a pointer to the authentication token is stored in the location pointed to by the *authtok* argument.

The *item* argument must have one of the following values:

PAM_AUTHTOK — Returns the current authentication token, or the new token when changing authentication tokens.

PAM_OLDAUTHTOK — Returns the previous authentication token when changing authentication tokens.

The *prompt* argument specifies a prompt to use if no token is cached. If it is NULL, the PAM_AUTHTOK_PROMPT or PAM_OLDAUTHTOK_PROMPT item, as appropriate, will be used. If that item is also NULL, a hardcoded default prompt will be used.

If *item* is set to PAM_AUTHTOK and there is a non-null PAM_OLDAUTHTOK item, **pam_get_authtok** will ask the user to confirm the new token by retyping it. If there is a mismatch, **pam_get_authtok** will return PAM_TRY_AGAIN.

RETURN VALUES

The **pam_get_authtok** function returns one of the following values:

[PAM_BUF_ERR] — Memory buffer error.

[PAM_CONV_ERR] — Conversation failure.

[PAM_SYSTEM_ERR] — System error.

[PAM_TRY_AGAIN] — Try again.

SEE ALSO

pam(3), pam_get_item(3), pam_get_user(3), pam_strerror(3)

STANDARDS

The **pam_get_authtok** function is an OpenPAM extension.

AUTHORS

The **pam_get_authtok** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_get_data — get module information

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_get_data(const pam_handle_t *pamh, const char *module_data_name,
    const void **data);
```

DESCRIPTION

The **pam_get_data** function looks up the opaque object associated with the string specified by the *module_data_name* argument, in the PAM context specified by the *pamh* argument. A pointer to the object is stored in the location pointed to by the *data* argument.

This function and its counterpart pam_set_data(3) are useful for managing data that are meaningful only to a particular service module.

RETURN VALUES

The **pam_get_data** function returns one of the following values:

[PAM_NO_MODULE_DATA]
: Module data not found.

[PAM_SYSTEM_ERR]
: System error.

SEE ALSO

pam(3), pam_set_data(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_get_data** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_get_item — get PAM information

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_get_item(const pam_handle_t *pamh, int item_type, const void **item);
```

DESCRIPTION

The **pam_get_item** function stores a pointer to the item specified by the *item_type* argument in the location specified by the *item* argument. The item is retrieved from the PAM context specified by the *pamh* argument. The following item types are recognized:

PAM_SERVICE — The name of the requesting service.

PAM_USER — The name of the user the application is trying to authenticate.

PAM_TTY — The name of the current terminal.

PAM_RHOST — The name of the applicant's host.

PAM_CONV — A *struct pam_conv* describing the current conversation function.

PAM_AUTHTOK — The current authentication token.

PAM_OLDAUTHTOK — The expired authentication token.

PAM_RUSER — The name of the applicant.

PAM_USER_PROMPT — The prompt to use when asking the applicant for a user name to authenticate as.

PAM_AUTHTOK_PROMPT — The prompt to use when asking the applicant for an authentication token.

PAM_OLDAUTHTOK_PROMPT — The prompt to use when asking the applicant for an expired authentication token prior to changing it.

PAM_SOCKADDR — The sockaddr_storage of the applicants's host.

PAM_NUSER — The "nested" user if this is a login on top of a previous one.

See pam_start(3) for a description of *struct pam_conv*.

RETURN VALUES

The **pam_get_item** function returns one of the following values:

[PAM_SYMBOL_ERR] — Invalid symbol.

[PAM_SYSTEM_ERR] — System error.

SEE ALSO

pam(3), pam_set_item(3), pam_start(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_get_item** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_get_user — retrieve user name

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_get_user(pam_handle_t *pamh, const char **user, const char *prompt);
```

DESCRIPTION

The **pam_get_user** function returns the name of the target user, as specified to pam_start(3). If no user was specified, nor set using pam_set_item(3), **pam_get_user** will prompt for a user name. Either way, a pointer to the user name is stored in the location pointed to by the *user* argument.

The *prompt* argument specifies a prompt to use if no user name is cached. If it is NULL, the PAM_USER_PROMPT will be used. If that item is also NULL, a hardcoded default prompt will be used.

RETURN VALUES

The **pam_get_user** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_get_authtok(3), pam_get_item(3), pam_set_item(3), pam_start(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_get_user** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_getenv — retrieve the value of a PAM environment variable

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

const char *
pam_getenv(pam_handle_t *pamh, const char *name);
```

DESCRIPTION

The **pam_getenv** function returns the value of an environment variable. Its semantics are similar to those of getenv(3), but it accesses the PAM context's environment list instead of the application's.

RETURN VALUES

The **pam_getenv** function returns NULL on failure.

SEE ALSO

getenv(3), pam(3), pam_getenvlist(3), pam_putenv(3), pam_setenv(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_getenv** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_getenvlist — returns a list of all the PAM environment variables

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

char **
pam_getenvlist(pam_handle_t *pamh);
```

DESCRIPTION

The **pam_getenvlist** function returns a copy of the given PAM context's environment list as a pointer to an array of strings. The last element in the array is `NULL`. The pointer is suitable for assignment to *environ*.

The array and the strings it lists are allocated using `malloc`(3), and should be released using `free`(3) after use:

```
char **envlist, **env;

envlist = environ;
environ = pam_getenvlist(pamh);
/* do something nifty */
for (env = environ; *env != NULL; env++)
    free(*env);
free(environ);
environ = envlist;
```

RETURN VALUES

The **pam_getenvlist** function returns `NULL` on failure.

SEE ALSO

`free`(3), `malloc`(3), `pam`(3), `pam_getenv`(3), `pam_putenv`(3), `pam_setenv`(3), `environ`(7)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_getenvlist** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_info — display an information message

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_info(const pam_handle_t *pamh, const char *fmt, ...);
```

DESCRIPTION

The **pam_info** function displays an informational message through the intermediary of the given PAM context's conversation function.

RETURN VALUES

The **pam_info** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_error(3), pam_prompt(3), pam_strerror(3), pam_vinfo(3)

STANDARDS

The **pam_info** function is an OpenPAM extension.

AUTHORS

The **pam_info** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_open_session — open a user session

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_open_session(pam_handle_t *pamh, int flags);
```

DESCRIPTION

The **pam_open_session** sets up a user session for a previously authenticated user. The session should later be torn down by a call to pam_close_session(3).

The *flags* argument is the binary or of zero or more of the following values:

PAM_SILENT Do not emit any messages.

If any other bits are set, **pam_open_session** will return PAM_SYMBOL_ERR.

RETURN VALUES

The **pam_open_session** function returns one of the following values:

[PAM_ABORT] General failure.

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_PERM_DENIED] Permission denied.

[PAM_SERVICE_ERR] Error in service module.

[PAM_SESSION_ERR] Session failure.

[PAM_SYMBOL_ERR] Invalid symbol.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_close_session(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_open_session** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_prompt — call the conversation function

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_prompt(const pam_handle_t *pamh, int style, char **resp,
    const char *fmt, ...);
```

DESCRIPTION

The **pam_prompt** function constructs a message from the specified format string and arguments and passes it to the given PAM context's conversation function.

A pointer to the response, or `NULL` if the conversation function did not return one, is stored in the location pointed to by the *resp* argument.

See `pam_vprompt`(3) for further details.

RETURN VALUES

The **pam_prompt** function returns one of the following values:

`[PAM_BUF_ERR]` Memory buffer error.

`[PAM_CONV_ERR]` Conversation failure.

`[PAM_SYSTEM_ERR]` System error.

SEE ALSO

`pam`(3), `pam_error`(3), `pam_info`(3), `pam_strerror`(3), `pam_vprompt`(3)

STANDARDS

The **pam_prompt** function is an OpenPAM extension.

AUTHORS

The **pam_prompt** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_putenv — set the value of an environment variable

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_putenv(pam_handle_t *pamh, const char *namevalue);
```

DESCRIPTION

The **pam_putenv** function sets a environment variable. Its semantics are similar to those of putenv(3), but it modifies the PAM context's environment list instead of the application's.

RETURN VALUES

The **pam_putenv** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_getenv(3), pam_getenvlist(3), pam_setenv(3), pam_strerror(3), putenv(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_putenv** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_set_data — set module information

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_set_data(pam_handle_t *pamh, const char *module_data_name, void *data,
    void (*cleanup)(pam_handle_t *pamh, void *data, int pam_end_status));
```

DESCRIPTION

The **pam_set_data** function associates a pointer to an opaque object with an arbitrary string specified by the *module_data_name* argument, in the PAM context specified by the *pamh* argument.

If not NULL, the *cleanup* argument should point to a function responsible for releasing the resources associated with the object.

This function and its counterpart pam_get_data(3) are useful for managing data that are meaningful only to a particular service module.

RETURN VALUES

The **pam_set_data** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

openpam_free_data(3), pam(3), pam_get_data(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_set_data** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_set_item — set authentication information

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_set_item(pam_handle_t *pamh, int item_type, const void *item);
```

DESCRIPTION

The **pam_set_item** function sets the item specified by the *item_type* argument to a copy of the object pointed to by the *item* argument. The item is stored in the PAM context specified by the *pamh* argument. See pam_get_item(3) for a list of recognized item types.

RETURN VALUES

The **pam_set_item** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_SYMBOL_ERR]
Invalid symbol.

[PAM_SYSTEM_ERR]
System error.

SEE ALSO

pam(3), pam_get_item(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_set_item** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_setcred — modify / delete user credentials for an authentication service

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_setcred(pam_handle_t *pamh, int flags);
```

DESCRIPTION

The **pam_setcred** function manages the application's credentials.

The *flags* argument is the binary or of zero or more of the following values:

PAM_SILENT
: Do not emit any messages.

PAM_ESTABLISH_CRED
: Establish the credentials of the target user.

PAM_DELETE_CRED
: Revoke all established credentials.

PAM_REINITIALIZE_CRED
: Fully reinitialise credentials.

PAM_REFRESH_CRED
: Refresh credentials.

The latter four are mutually exclusive.

If any other bits are set, **pam_setcred** will return PAM_SYMBOL_ERR.

RETURN VALUES

The **pam_setcred** function returns one of the following values:

[PAM_ABORT]
: General failure.

[PAM_BUF_ERR]
: Memory buffer error.

[PAM_CONV_ERR]
: Conversation failure.

[PAM_CRED_ERR]
: Failed to set user credentials.

[PAM_CRED_EXPIRED]
: User credentials have expired.

[PAM_CRED_UNAVAIL]
: Failed to retrieve user credentials.

[PAM_PERM_DENIED]
: Permission denied.

[PAM_SERVICE_ERR]
: Error in service module.

[PAM_SYMBOL_ERR]
: Invalid symbol.

[PAM_SYSTEM_ERR]
System error.

[PAM_USER_UNKNOWN]
Unknown user.

SEE ALSO

pam(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_setcred** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_setenv — mirrors setenv(3)

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_setenv(pam_handle_t *pamh, const char *name, const char *value,
    int overwrite);
```

DESCRIPTION

The **pam_setenv** function sets a environment variable. Its semantics are similar to those of setenv(3), but it modifies the PAM context's environment list instead of the application's.

RETURN VALUES

The **pam_setenv** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_getenv(3), pam_getenvlist(3), pam_putenv(3), pam_strerror(3), setenv(3)

STANDARDS

The **pam_setenv** function is an OpenPAM extension.

AUTHORS

The **pam_setenv** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_sm_acct_mgmt — service module implementation for pam_acct_mgmt

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int
pam_sm_acct_mgmt(pam_handle_t *pamh, int flags, int argc,
    const char **argv);
```

DESCRIPTION

The **pam_sm_acct_mgmt** function is the service module's implementation of the pam_acct_mgmt(3) API function.

RETURN VALUES

The **pam_sm_acct_mgmt** function returns one of the following values:

[PAM_ABORT] General failure.

[PAM_ACCT_EXPIRED]
User account has expired.

[PAM_AUTH_ERR] Authentication error.

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_IGNORE] Ignore this module.

[PAM_NEW_AUTHTOK_REQD]
New authentication token required.

[PAM_PERM_DENIED]
Permission denied.

[PAM_SERVICE_ERR]
Error in service module.

[PAM_SYSTEM_ERR]
System error.

[PAM_USER_UNKNOWN]
Unknown user.

SEE ALSO

pam(3), pam_acct_mgmt(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_sm_acct_mgmt** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_sm_authenticate — service module implementation for pam_authenticate

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int
pam_sm_authenticate(pam_handle_t *pamh, int flags, int argc,
    const char **argv);
```

DESCRIPTION

The **pam_sm_authenticate** function is the service module's implementation of the pam_authenticate(3) API function.

RETURN VALUES

The **pam_sm_authenticate** function returns one of the following values:

[PAM_ABORT] General failure.

[PAM_AUTHINFO_UNAVAIL]
Authentication information is unavailable.

[PAM_AUTH_ERR] Authentication error.

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_CRED_INSUFFICIENT]
Insufficient credentials.

[PAM_IGNORE] Ignore this module.

[PAM_MAXTRIES] Maximum number of tries exceeded.

[PAM_PERM_DENIED]
Permission denied.

[PAM_SERVICE_ERR]
Error in service module.

[PAM_SYSTEM_ERR]
System error.

[PAM_USER_UNKNOWN]
Unknown user.

SEE ALSO

pam(3), pam_authenticate(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_sm_authenticate** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_sm_chauthtok — service module implementation for pam_chauthtok

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int
pam_sm_chauthtok(pam_handle_t *pamh, int flags, int argc,
    const char **argv);
```

DESCRIPTION

The **pam_sm_chauthtok** function is the service module's implementation of the pam_chauthtok(3) API function.

RETURN VALUES

The **pam_sm_chauthtok** function returns one of the following values:

[PAM_ABORT] General failure.

[PAM_AUTHTOK_DISABLE_AGING]
Authentication token aging disabled.

[PAM_AUTHTOK_ERR]
Authentication token failure.

[PAM_AUTHTOK_LOCK_BUSY]
Authentication token lock busy.

[PAM_AUTHTOK_RECOVERY_ERR]
Failed to recover old authentication token.

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_IGNORE] Ignore this module.

[PAM_PERM_DENIED]
Permission denied.

[PAM_SERVICE_ERR]
Error in service module.

[PAM_SYSTEM_ERR]
System error.

[PAM_TRY_AGAIN] Try again.

SEE ALSO

pam(3), pam_chauthtok(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_sm_chauthtok** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_sm_close_session — service module implementation for pam_close_session

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int
pam_sm_close_session(pam_handle_t *pamh, int flags, int args,
    const char **argv);
```

DESCRIPTION

The **pam_sm_close_session** function is the service module's implementation of the pam_close_session(3) API function.

RETURN VALUES

The **pam_sm_close_session** function returns one of the following values:

[PAM_ABORT] General failure.

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_IGNORE] Ignore this module.

[PAM_PERM_DENIED] Permission denied.

[PAM_SERVICE_ERR] Error in service module.

[PAM_SESSION_ERR] Session failure.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_close_session(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_sm_close_session** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_sm_open_session — service module implementation for pam_open_session

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int
pam_sm_open_session(pam_handle_t *pamh, int flags, int argc,
    const char **argv);
```

DESCRIPTION

The **pam_sm_open_session** function is the service module's implementation of the pam_open_session(3) API function.

RETURN VALUES

The **pam_sm_open_session** function returns one of the following values:

[PAM_ABORT] General failure.

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_IGNORE] Ignore this module.

[PAM_PERM_DENIED] Permission denied.

[PAM_SERVICE_ERR] Error in service module.

[PAM_SESSION_ERR] Session failure.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_open_session(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_sm_open_session** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_sm_setcred — service module implementation for pam_setcred

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int
pam_sm_setcred(pam_handle_t *pamh, int flags, int argc, const char **argv);
```

DESCRIPTION

The **pam_sm_setcred** function is the service module's implementation of the pam_setcred(3) API function.

RETURN VALUES

The **pam_sm_setcred** function returns one of the following values:

[PAM_ABORT] General failure.

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_CRED_ERR] Failed to set user credentials.

[PAM_CRED_EXPIRED] User credentials have expired.

[PAM_CRED_UNAVAIL] Failed to retrieve user credentials.

[PAM_IGNORE] Ignore this module.

[PAM_PERM_DENIED] Permission denied.

[PAM_SERVICE_ERR] Error in service module.

[PAM_SYSTEM_ERR] System error.

[PAM_USER_UNKNOWN] Unknown user.

SEE ALSO

pam(3), pam_setcred(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_sm_setcred** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_start — initiate a PAM transaction

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_start(const char *service, const char *user,
    const struct pam_conv *pam_conv, pam_handle_t **pamh);
```

DESCRIPTION

The **pam_start** function creates and initializes a PAM context.

The *service* argument specifies the name of the policy to apply, and is stored in the PAM_SERVICE item in the created context.

The *user* argument specifies the name of the target user - the user the created context will serve to authenticate. It is stored in the PAM_USER item in the created context.

The *pam_conv* argument points to a *struct pam_conv* describing the conversation function to use; see *pam_conv* for details.

RETURN VALUES

The **pam_start** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_end(3), pam_get_item(3), pam_set_item(3), pam_strerror(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_start** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_strerror — get PAM standard error message string

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

const char *
pam_strerror(const pam_handle_t *pamh, int error_number);
```

DESCRIPTION

The **pam_strerror** function returns a pointer to a string containing a textual description of the error indicated by the *error_number* argument, in the context of the PAM transaction described by the *pamh* argument.

RETURN VALUES

The **pam_strerror** function returns NULL on failure.

SEE ALSO

pam(3)

STANDARDS

X/Open Single Sign-On Service (XSSO) - Pluggable Authentication Modules, June 1997.

AUTHORS

The **pam_strerror** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_verror — display an error message

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_verror(const pam_handle_t *pamh, const char *fmt, va_list ap);
```

DESCRIPTION

The **pam_verror** function passes its arguments to pam_vprompt(3) with a style argument of PAM_ERROR_MSG, and discards the response.

RETURN VALUES

The **pam_verror** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_error(3), pam_strerror(3), pam_vinfo(3), pam_vprompt(3)

STANDARDS

The **pam_verror** function is an OpenPAM extension.

AUTHORS

The **pam_verror** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_vinfo — display an information message

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_vinfo(const pam_handle_t *pamh, const char *fmt, va_list ap);
```

DESCRIPTION

The **pam_vinfo** function passes its arguments to pam_vprompt(3) with a style argument of PAM_TEXT_INFO, and discards the response.

RETURN VALUES

The **pam_vinfo** function returns one of the following values:

[PAM_BUF_ERR] Memory buffer error.

[PAM_CONV_ERR] Conversation failure.

[PAM_SYSTEM_ERR] System error.

SEE ALSO

pam(3), pam_info(3), pam_strerror(3), pam_verror(3), pam_vprompt(3)

STANDARDS

The **pam_vinfo** function is an OpenPAM extension.

AUTHORS

The **pam_vinfo** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

NAME

pam_vprompt — call the conversation function

LIBRARY

Pluggable Authentication Module Library (libpam, −lpam)

SYNOPSIS

```
#include <sys/types.h>
#include <security/pam_appl.h>

int
pam_vprompt(const pam_handle_t *pamh, int style, char **resp,
    const char *fmt, va_list ap);
```

DESCRIPTION

The **pam_vprompt** function constructs a string from the *fmt* and *ap* arguments using vsnprintf(3), and passes it to the given PAM context's conversation function.

The *style* argument specifies the type of interaction requested, and must be one of the following:

PAM_PROMPT_ECHO_OFF
: Display the message and obtain the user's response without displaying it.

PAM_PROMPT_ECHO_ON
: Display the message and obtain the user's response.

PAM_ERROR_MSG
: Display the message as an error message, and do not wait for a response.

PAM_TEXT_INFO
: Display the message as an informational message, and do not wait for a response.

A pointer to the response, or NULL if the conversation function did not return one, is stored in the location pointed to by the *resp* argument.

The message and response should not exceed PAM_MAX_MSG_SIZE or PAM_MAX_RESP_SIZE, respectively. If they do, they may be truncated.

RETURN VALUES

The **pam_vprompt** function returns one of the following values:

[PAM_BUF_ERR]
: Memory buffer error.

[PAM_CONV_ERR]
: Conversation failure.

[PAM_SYSTEM_ERR]
: System error.

SEE ALSO

pam(3), pam_error(3), pam_info(3), pam_prompt(3), pam_strerror(3), pam_verror(3), pam_vinfo(3), vsnprintf(3)

STANDARDS

The **pam_vprompt** function is an OpenPAM extension.

AUTHORS

The **pam_vprompt** function and this manual page were developed for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research pro-

gram.

NAME

parse_time, **print_time_table**, **unparse_time**, **unparse_time_approx**, — parse and unparse time intervals

LIBRARY

The roken library (libroken, -lroken)

SYNOPSIS

```
#include <parse_time.h>

int
parse_time(const char *timespec, const char *def_unit);

void
print_time_table(FILE *f);

size_t
unparse_time(int seconds, char *buf, size_t len);

size_t
unparse_time_approx(int seconds, char *buf, size_t len);
```

DESCRIPTION

The **parse_time**() function converts a the period of time specified in into a number of seconds. The *timespec* can be any number of ⟨number unit⟩ pairs separated by comma and whitespace. The number can be negative. Number without explicit units are taken as being *def_unit*.

The **unparse_time**() and **unparse_time_approx**() does the opposite of **parse_time**(), that is they take a number of seconds and express that as human readable string. *unparse_time* produces an exact time, while *unparse_time_approx* restricts the result to only include one units.

print_time_table() prints a descriptive list of available units on the passed file descriptor.

The possible units include:

second, s	
minute, m	
hour, h	
day	
week	seven days
month	30 days
year	365 days

Units names can be arbitrarily abbreviated (as long as they are unique).

RETURN VALUES

parse_time() returns the number of seconds that represents the expression in *timespec* or -1 on error. **unparse_time**() and **unparse_time_approx**() return the number of characters written to *buf*. if the return value is greater than or equal to the *len* argument, the string was too short and some of the printed characters were discarded.

EXAMPLES

```
#include <stdio.h>
#include <parse_time.h>

int
main(int argc, char **argv)
```

```
{
    int i;
    int result;
    char buf[128];
    print_time_table(stdout);
    for (i = 1; i < argc; i++) {
        result = parse_time(argv[i], "second");
        if(result == -1) {
            fprintf(stderr, "%s: parse error\n", argv[i]);
            continue;
        }
        printf("--\n");
        printf("parse_time = %d\n", result);
        unparse_time(result, buf, sizeof(buf));
        printf("unparse_time = %s\n", buf);
        unparse_time_approx(result, buf, sizeof(buf));
        printf("unparse_time_approx = %s\n", buf);
    }
    return 0;
}
```

```
$ ./a.out "1 minute 30 seconds" "90 s" "1 y -1 s"
1   year = 365 days
1  month = 30 days
1   week = 7 days
1    day = 24 hours
1   hour = 60 minutes
1 minute = 60 seconds
1 second
--
parse_time = 90
unparse_time = 1 minute 30 seconds
unparse_time_approx = 1 minute
--
parse_time = 90
unparse_time = 1 minute 30 seconds
unparse_time_approx = 1 minute
--
parse_time = 31535999
unparse_time = 12 months 4 days 23 hours 59 minutes 59 seconds
unparse_time_approx = 12 months
```

BUGS

Since **parse_time**() returns -1 on error there is no way to parse "minus one second". Currently "s" at the end of units is ignored. This is a hack for English plural forms. If these functions are ever localised, this scheme will have to change.

NAME

parsedate — date parsing function

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

time_t
parsedate(const char *datestr, const time_t *time, const int *tzoff);
```

DESCRIPTION

The **parsedate** function parses a datetime from *datestr* described in english relative to an optional *time* point and an optional timezone offset in seconds specified in *tzoff*. If either *time* or *tzoff* are NULL, then the current time and timezone offset are used.

The *datestr* is a sequence of white-space separated items. The white-space is optional the concatenated items are not ambiguous. An empty *datestr* is equivalent to midnight today (the beginning of this day).

The following words have the indicated numeric meanings: last = −1, this = 0, first or next 1, second is unused so that it is not confused with “seconds”, third = 3, fourth = 4, fifth = 5, sixth = 6, seventh = 7, eighth = 8, ninth = 9, tenth = 10, eleventh = 11, twelfth = 12.

The following words are recognized in English only: AM, PM, a.m., p.m.

The months: january, february, march, april, may, june, july, august, september, sept, october, november, december,

The days of the week: sunday, monday, tuesday, tues, wednesday, wednes, thursday, thur, thurs, friday, saturday.

Time units: year, month, fortnight, week, day, hour, minute, min, second, sec, tomorrow, yesterday.

Timezone names: gmt, ut, utc, wet, bst, wat, at, ast, adt, est, edt, cst, cdt, mst, mdt, pst, pdt, yst, ydt, hst, hdt, cat, ahst, nt, idlw, cet, met, mewt, mest, swt, sst, fwt, fst, eet, bt, zp4, zp5, zp6, wast, wadt, cct, jst, east, eadt, gst, nzt, nzst, nzdt, idle.

A variety of unambiguous dates are recognized:

69-09-10	For years between 69-99 we assume 1900+ and for years between 0-68 we assume 2000+.
2006-11-17	An ISO-8601 date.
10/1/2000	October 10, 2000; the common US format.
20 Jun 1994	
23jun2001	
1-sep-06	Other common abbreviations.
1/11	the year can be omitted

As well as times:
10:01
10:12pm
12:11:01.000012
12:21-0500

Relative items are also supported:

-1 month
last friday
one week ago
this thursday
next sunday
+2 years

RETURN VALUES

parsedate() returns the number of seconds passed since the Epoch, or `-1` if the date could not be parsed properly.

SEE ALSO

`date`(1), `eeprom`(8)

HISTORY

The parser used in **parsedate**() was originally written by Steven M. Bellovin while at the University of North Carolina at Chapel Hill. It was later tweaked by a couple of people on Usenet. Completely overhauled by Rich $alz and Jim Berets in August, 1990.

The **parsedate**() function first appeared in NetBSD 4.0.

NAME

pause — stop until signal

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
pause(void);
```

DESCRIPTION

Pause is made obsolete by `sigsuspend`**(2).**

The **pause**() function forces a process to pause until a signal is received from either the `kill`(2) function or an interval timer. (See `setitimer`(2).) Upon termination of a signal handler started during a **pause**(), the **pause**() call will return.

RETURN VALUES

Always returns −1.

ERRORS

The **pause**() function always returns:

`[EINTR]` The call was interrupted.

SEE ALSO

`kill`(2), `poll`(2), `select`(2), `setitimer`(2), `sigsuspend`(2)

STANDARDS

The **pause**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

A **pause**() syscall appeared in Version 6 AT&T UNIX.

NAME

pcap − Packet Capture library

SYNOPSIS

```
#include <pcap.h>

char errbuf[PCAP_ERRBUF_SIZE];

pcap_t *pcap_open_live(const char *device, int snaplen,
                       int promisc, int to_ms, char *errbuf)
pcap_t *pcap_open_dead(int linktype, int snaplen)
pcap_t *pcap_open_offline(const char *fname, char *errbuf)
pcap_t *pcap_fopen_offline(FILE *fp, char *errbuf)
pcap_dumper_t *pcap_dump_open(pcap_t *p, const char *fname)
pcap_dumper_t *pcap_dump_fopen(pcap_t *p, FILE *fp)

int pcap_setnonblock(pcap_t *p, int nonblock, char *errbuf);
int pcap_getnonblock(pcap_t *p, char *errbuf);

int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)
void pcap_freealldevs(pcap_if_t *alldevs)
char *pcap_lookupdev(char *errbuf)
int pcap_lookupnet(const char *device, bpf_u_int32 *netp,
                       bpf_u_int32 *maskp, char *errbuf)

typedef void (*pcap_handler)(u_char *user, const struct pcap_pkthdr *h,
                                                   const u_char *bytes);
int pcap_dispatch(pcap_t *p, int cnt,
                       pcap_handler callback, u_char *user)
int pcap_loop(pcap_t *p, int cnt,
                       pcap_handler callback, u_char *user)
void pcap_dump(u_char *user, struct pcap_pkthdr *h,
                       u_char *sp)

int pcap_compile(pcap_t *p, struct bpf_program *fp,
                       const char *str, int optimize, bpf_u_int32 netmask)
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
void pcap_freecode(struct bpf_program *)
int pcap_setdirection(pcap_t *p, pcap_direction_t d)

const u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
int pcap_next_ex(pcap_t *p, struct pcap_pkthdr **pkt_header,
                       const u_char **pkt_data)

void pcap_breakloop(pcap_t *)

int pcap_inject(pcap_t *p, const void *buf, size_t size)
int pcap_sendpacket(pcap_t *p, const u_char *buf, int size)

int pcap_datalink(pcap_t *p)
int pcap_list_datalinks(pcap_t *p, int **dlt_buf);
int pcap_set_datalink(pcap_t *p, int dlt);
int pcap_datalink_name_to_val(const char *name);
const char *pcap_datalink_val_to_name(int dlt);
const char *pcap_datalink_val_to_description(int dlt);
int pcap_snapshot(pcap_t *p)
int pcap_is_swapped(pcap_t *p)
int pcap_major_version(pcap_t *p)
int pcap_minor_version(pcap_t *p)
int pcap_stats(pcap_t *p, struct pcap_stat *ps)
FILE *pcap_file(pcap_t *p)
```

int pcap_fileno(pcap_t *p)
int pcap_get_selectable_fd(pcap_t *p);
void pcap_perror(pcap_t *p, char *prefix)
char *pcap_geterr(pcap_t *p)
const char *pcap_strerror(int error)
const char *pcap_lib_version(void)

void pcap_close(pcap_t *p)
int pcap_dump_flush(pcap_dumper_t *p)
long pcap_dump_ftell(pcap_dumper_t *p)
FILE *pcap_dump_file(pcap_dumper_t *p)
void pcap_dump_close(pcap_dumper_t *p)

DESCRIPTION

The Packet Capture library provides a high level interface to packet capture systems. All packets on the network, even those destined for other hosts, are accessible through this mechanism.

ROUTINES

NOTE: *errbuf* in **pcap_open_live()**, **pcap_open_dead()**, **pcap_open_offline()**, **pcap_fopen_offline()**, **pcap_setnonblock()**, **pcap_getnonblock()**, **pcap_findalldevs()**, **pcap_lookupdev()**, and **pcap_lookupnet()** is assumed to be able to hold at least **PCAP_ERRBUF_SIZE** chars.

pcap_open_live() is used to obtain a packet capture descriptor to look at packets on the network. *device* is a string that specifies the network device to open; on Linux systems with 2.2 or later kernels, a *device* argument of "any" or **NULL** can be used to capture packets from all interfaces. *snaplen* specifies the maximum number of bytes to capture. If this value is less than the size of a packet that is captured, only the first *snaplen* bytes of that packet will be captured and provided as packet data. A value of 65535 should be sufficient, on most if not all networks, to capture all the data available from the packet. *promisc* specifies if the interface is to be put into promiscuous mode. (Note that even if this parameter is false, the interface could well be in promiscuous mode for some other reason.) For now, this doesn't work on the "any" device; if an argument of "any" or NULL is supplied, the *promisc* flag is ignored. *to_ms* specifies the read timeout in milliseconds. The read timeout is used to arrange that the read not necessarily return immediately when a packet is seen, but that it wait for some amount of time to allow more packets to arrive and to read multiple packets from the OS kernel in one operation. Not all platforms support a read timeout; on platforms that don't, the read timeout is ignored. A zero value for *to_ms*, on platforms that support a read timeout, will cause a read to wait forever to allow enough packets to arrive, with no timeout. *errbuf* is used to return error or warning text. It will be set to error text when **pcap_open_live()** fails and returns **NULL**. *errbuf* may also be set to warning text when **pcap_open_live()** succeds; to detect this case the caller should store a zero-length string in *errbuf* before calling **pcap_open_live()** and display the warning to the user if *errbuf* is no longer a zero-length string.

pcap_open_dead() is used for creating a **pcap_t** structure to use when calling the other functions in libpcap. It is typically used when just using libpcap for compiling BPF code.

pcap_open_offline() is called to open a ''savefile'' for reading. *fname* specifies the name of the file to open. The file has the same format as those used by **tcpdump(1)** and **tcpslice(1)**. The name "-" in a synonym for **stdin**. Alternatively, you may call **pcap_fopen_offline()** to read dumped data from an existing open stream *fp*. Note that on Windows, that stream should be opened in binary mode. *errbuf* is used to return error text and is only set when **pcap_open_offline()** or **pcap_fopen_offline()** fails and returns **NULL**.

pcap_dump_open() is called to open a ''savefile'' for writing. The name "-" in a synonym for **stdout**. **NULL** is returned on failure. *p* is a *pcap* struct as returned by **pcap_open_offline()** or **pcap_open_live()**. *fname* specifies the name of the file to open. Alternatively, you may call **pcap_dump_fopen()** to write data to an existing open stream *fp*. Note that on Windows, that stream should be opened in binary mode. If **NULL** is returned, **pcap_geterr()** can be used to get the error text.

pcap_setnonblock() puts a capture descriptor, opened with **pcap_open_live()**, into ''non-blocking'' mode,

or takes it out of ‘‘non-blocking’’ mode, depending on whether the *nonblock* argument is non-zero or zero. It has no effect on ‘‘savefiles’’. If there is an error, −1 is returned and *errbuf* is filled in with an appropriate error message; otherwise, 0 is returned. In ‘‘non-blocking’’ mode, an attempt to read from the capture descriptor with **pcap_dispatch()** will, if no packets are currently available to be read, return 0 immediately rather than blocking waiting for packets to arrive. **pcap_loop()** and **pcap_next()** will not work in ‘‘non-blocking’’ mode.

pcap_getnonblock() returns the current ‘‘non-blocking’’ state of the capture descriptor; it always returns 0 on ‘‘savefiles’’. If there is an error, −1 is returned and *errbuf* is filled in with an appropriate error message.

pcap_findalldevs() constructs a list of network devices that can be opened with **pcap_open_live()**. (Note that there may be network devices that cannot be opened with **pcap_open_live()** by the process calling **pcap_findalldevs()**, because, for example, that process might not have sufficient privileges to open them for capturing; if so, those devices will not appear on the list.) *alldevsp* is set to point to the first element of the list; each element of the list is of type **pcap_if_t**, and has the following members:

next if not **NULL**, a pointer to the next element in the list; **NULL** for the last element of the list

name a pointer to a string giving a name for the device to pass to **pcap_open_live()**

description
if not **NULL**, a pointer to a string giving a human-readable description of the device

addresses
a pointer to the first element of a list of addresses for the interface

flags interface flags:

PCAP_IF_LOOPBACK
set if the interface is a loopback interface

Each element of the list of addresses is of type **pcap_addr_t**, and has the following members:

next if not **NULL**, a pointer to the next element in the list; **NULL** for the last element of the list

addr a pointer to a **struct sockaddr** containing an address

netmask
if not **NULL**, a pointer to a **struct sockaddr** that contains the netmask corresponding to the address pointed to by **addr**

broadaddr
if not **NULL**, a pointer to a **struct sockaddr** that contains the broadcast address corresponding to the address pointed to by **addr**; may be null if the interface doesn’t support broadcasts

dstaddr
if not **NULL**, a pointer to a **struct sockaddr** that contains the destination address corresponding to the address pointed to by **addr**; may be null if the interface isn’t a point-to-point interface

Note that not all the addresses in the list of addresses are necessarily IPv4 or IPv6 addresses - you must check the **sa_family** member of the **struct sockaddr** before interpreting the contents of the address.

−1 is returned on failure, in which case **errbuf** is filled in with an appropriate error message; **0** is returned on success.

pcap_freealldevs() is used to free a list allocated by **pcap_findalldevs()**.

pcap_lookupdev() returns a pointer to a network device suitable for use with **pcap_open_live()** and **pcap_lookupnet()**. If there is an error, **NULL** is returned and *errbuf* is filled in with an appropriate error message.

pcap_lookupnet() is used to determine the network number and mask associated with the network device

device. Both *netp* and *maskp* are *bpf_u_int32* pointers. A return of −1 indicates an error in which case *errbuf* is filled in with an appropriate error message.

pcap_dispatch() is used to collect and process packets. *cnt* specifies the maximum number of packets to process before returning. This is not a minimum number; when reading a live capture, only one bufferful of packets is read at a time, so fewer than *cnt* packets may be processed. A *cnt* of −1 processes all the packets received in one buffer when reading a live capture, or all the packets in the file when reading a ''savefile''. *callback* specifies a routine to be called with three arguments: a *u_char* pointer which is passed in from **pcap_dispatch()**, a *const struct pcap_pkthdr* pointer to a structure with the following members:

ts a *struct timeval* containing the time when the packet was captured

caplen a *bpf_u_int32* giving the number of bytes of the packet that are available from the capture

len a *bpf_u_int32* giving the length of the packet, in bytes (which might be more than the number of bytes available from the capture, if the length of the packet is larger than the maximum number of bytes to capture)

and a *const u_char* pointer to the first **caplen** (as given in the *struct pcap_pkthdr* a pointer to which is passed to the callback routine) bytes of data from the packet (which won't necessarily be the entire packet; to capture the entire packet, you will have to provide a value for *snaplen* in your call to **pcap_open_live()** that is sufficiently large to get all of the packet's data - a value of 65535 should be sufficient on most if not all networks).

The number of packets read is returned. 0 is returned if no packets were read from a live capture (if, for example, they were discarded because they didn't pass the packet filter, or if, on platforms that support a read timeout that starts before any packets arrive, the timeout expires before any packets arrive, or if the file descriptor for the capture device is in non-blocking mode and no packets were available to be read) or if no more packets are available in a ''savefile.'' A return of −1 indicates an error in which case **pcap_perror()** or **pcap_geterr()** may be used to display the error text. A return of −2 indicates that the loop terminated due to a call to **pcap_breakloop()** before any packets were processed. **If your application uses pcap_breakloop(), make sure that you explicitly check for −1 and −2, rather than just checking for a return value < 0.**

NOTE: when reading a live capture, **pcap_dispatch()** will not necessarily return when the read times out; on some platforms, the read timeout isn't supported, and, on other platforms, the timer doesn't start until at least one packet arrives. This means that the read timeout should **NOT** be used in, for example, an interactive application, to allow the packet capture loop to ''poll'' for user input periodically, as there's no guarantee that **pcap_dispatch()** will return after the timeout expires.

pcap_loop() is similar to **pcap_dispatch()** except it keeps reading packets until *cnt* packets are processed or an error occurs. It does **not** return when live read timeouts occur. Rather, specifying a non-zero read timeout to **pcap_open_live()** and then calling **pcap_dispatch()** allows the reception and processing of any packets that arrive when the timeout occurs. A negative *cnt* causes **pcap_loop()** to loop forever (or at least until an error occurs). −1 is returned on an error; 0 is returned if *cnt* is exhausted; −2 is returned if the loop terminated due to a call to **pcap_breakloop()** before any packets were processed. **If your application uses pcap_breakloop(), make sure that you explicitly check for −1 and −2, rather than just checking for a return value < 0.**

pcap_next() reads the next packet (by calling **pcap_dispatch()** with a *cnt* of 1) and returns a *u_char* pointer to the data in that packet. (The *pcap_pkthdr* struct for that packet is not supplied.) **NULL** is returned if an error occured, or if no packets were read from a live capture (if, for example, they were discarded because they didn't pass the packet filter, or if, on platforms that support a read timeout that starts before any packets arrive, the timeout expires before any packets arrive, or if the file descriptor for the capture device is in non-blocking mode and no packets were available to be read), or if no more packets are available in a ''savefile.'' Unfortunately, there is no way to determine whether an error occured or not.

pcap_next_ex() reads the next packet and returns a success/failure indication:

1 the packet was read without problems

0 packets are being read from a live capture, and the timeout expired

−1 an error occurred while reading the packet

−2 packets are being read from a ‘‘savefile’’, and there are no more packets to read from the savefile.

If the packet was read without problems, the pointer pointed to by the *pkt_header* argument is set to point to the *pcap_pkthdr* struct for the packet, and the pointer pointed to by the *pkt_data* argument is set to point to the data in the packet.

pcap_breakloop() sets a flag that will force **pcap_dispatch()** or **pcap_loop()** to return rather than looping; they will return the number of packets that have been processed so far, or −2 if no packets have been processed so far.

This routine is safe to use inside a signal handler on UNIX or a console control handler on Windows, as it merely sets a flag that is checked within the loop.

The flag is checked in loops reading packets from the OS - a signal by itself will not necessarily terminate those loops - as well as in loops processing a set of packets returned by the OS. **Note that if you are catching signals on UNIX systems that support restarting system calls after a signal, and calling pcap_breakloop() in the signal handler, you must specify, when catching those signals, that system calls should NOT be restarted by that signal. Otherwise, if the signal interrupted a call reading packets in a live capture, when your signal handler returns after calling pcap_breakloop(), the call will be restarted, and the loop will not terminate until more packets arrive and the call completes.**

Note also that, in a multi-threaded application, if one thread is blocked in **pcap_dispatch()**, **pcap_loop()**, **pcap_next()**, or **pcap_next_ex()**, a call to **pcap_breakloop()** in a different thread will not unblock that thread; you will need to use whatever mechanism the OS provides for breaking a thread out of blocking calls in order to unblock the thread, such as thread cancellation in systems that support POSIX threads.

Note that **pcap_next()** will, on some platforms, loop reading packets from the OS; that loop will not necessarily be terminated by a signal, so **pcap_breakloop()** should be used to terminate packet processing even if **pcap_next()** is being used.

pcap_breakloop() does not guarantee that no further packets will be processed by **pcap_dispatch()** or **pcap_loop()** after it is called; at most one more packet might be processed.

If −2 is returned from **pcap_dispatch()** or **pcap_loop()**, the flag is cleared, so a subsequent call will resume reading packets. If a positive number is returned, the flag is not cleared, so a subsequent call will return −2 and clear the flag.

pcap_inject() sends a raw packet through the network interface; *buf* points to the data of the packet, including the link-layer header, and *size* is the number of bytes in the packet. It returns the number of bytes written on success. A return of −1 indicates an error in which case **pcap_perror()** or **pcap_geterr()** may be used to display the error text. Note that, even if you successfully open the network interface, you might not have permission to send packets on it, or it might not support sending packets; as *pcap_open_live()* doesn’t have a flag to indicate whether to open for capturing, sending, or capturing and sending, you cannot request an open that supports sending and be notified at open time whether sending will be possible. Note also that some devices might not support sending packets.

Note that, on some platforms, the link-layer header of the packet that’s sent might not be the same as the link-layer header of the packet supplied to **pcap_inject()**, as the source link-layer address, if the header contains such an address, might be changed to be the address assigned to the interface on which the packet it sent, if the platform doesn’t support sending completely raw and unchanged packets. Even worse, some drivers on some platforms might change the link-layer type field to whatever value libpcap used when attaching to the device, even on platforms that *do* nominally support sending completely raw and unchanged packets.

pcap_sendpacket() is like **pcap_inject()**, but it returns 0 on success and −1 on failure. (**pcap_inject()** comes from OpenBSD; **pcap_sendpacket()** comes from WinPcap. Both are provided for compatibility.)

pcap_dump() outputs a packet to the ‘‘savefile’’ opened with **pcap_dump_open()**. Note that its calling

arguments are suitable for use with **pcap_dispatch()** or **pcap_loop()**. If called directly, the *user* parameter is of type *pcap_dumper_t* as returned by **pcap_dump_open()**.

pcap_compile() is used to compile the string *str* into a filter program. *program* is a pointer to a *bpf_program* struct and is filled in by **pcap_compile()**. *optimize* controls whether optimization on the resulting code is performed. *netmask* specifies the IPv4 netmask of the network on which packets are being captured; it is used only when checking for IPv4 broadcast addresses in the filter program. If the netmask of the network on which packets are being captured isn't known to the program, or if packets are being captured on the Linux "any" pseudo-interface that can capture on more than one network, a value of 0 can be supplied; tests for IPv4 broadcast addreses won't be done correctly, but all other tests in the filter program will be OK. A return of −1 indicates an error in which case **pcap_geterr()** may be used to display the error text.

pcap_compile_nopcap() is similar to **pcap_compile()** except that instead of passing a pcap structure, one passes the snaplen and linktype explicitly. It is intended to be used for compiling filters for direct BPF usage, without necessarily having called **pcap_open()**. A return of −1 indicates an error; the error text is unavailable. (**pcap_compile_nopcap()** is a wrapper around **pcap_open_dead()**, **pcap_compile()**, and **pcap_close()**; the latter three routines can be used directly in order to get the error text for a compilation error.)

pcap_setfilter() is used to specify a filter program. *fp* is a pointer to a *bpf_program* struct, usually the result of a call to **pcap_compile()**. **−1** is returned on failure, in which case **pcap_geterr()** may be used to display the error text; **0** is returned on success.

pcap_freecode() is used to free up allocated memory pointed to by a *bpf_program* struct generated by **pcap_compile()** when that BPF program is no longer needed, for example after it has been made the filter program for a pcap structure by a call to **pcap_setfilter()**.

pcap_setdirection() is used to specify a direction that packets will be captured. *pcap_direction_t* is one of the constants **PCAP_D_IN**, **PCAP_D_OUT** or **PCAP_D_INOUT**. **PCAP_D_IN** will only capture packets received by the device, **PCAP_D_OUT** will only capture packets sent by the device and **PCAP_D_INOUT** will capture packets received by or sent by the device. **PCAP_D_INOUT** is the default setting if this function is not called. This isn't necessarily supported on all platforms; some platforms might return an error, and some other platforms might not support **PCAP_D_OUT**. This operation is not supported if a ''savefile'' is being read. **−1** is returned on failure, **0** is returned on success.

pcap_datalink() returns the link layer type; link layer types it can return include:

DLT_NULL
BSD loopback encapsulation; the link layer header is a 4-byte field, in *host* byte order, containing a PF_ value from **socket.h** for the network-layer protocol of the packet.

Note that ''host byte order'' is the byte order of the machine on which the packets are captured, and the PF_ values are for the OS of the machine on which the packets are captured; if a live capture is being done, ''host byte order'' is the byte order of the machine capturing the packets, and the PF_ values are those of the OS of the machine capturing the packets, but if a ''savefile'' is being read, the byte order and PF_ values are *not* necessarily those of the machine reading the capture file.

DLT_EN10MB
Ethernet (10Mb, 100Mb, 1000Mb, and up)

DLT_IEEE802
IEEE 802.5 Token Ring

DLT_ARCNET
ARCNET

DLT_SLIP
SLIP; the link layer header contains, in order:

a 1-byte flag, which is 0 for packets received by the machine and 1 for packets sent by

the machine;

a 1-byte field, the upper 4 bits of which indicate the type of packet, as per RFC 1144:

0x40 an unmodified IP datagram (TYPE_IP);

0x70 an uncompressed-TCP IP datagram (UNCOMPRESSED_TCP), with that byte being the first byte of the raw IP header on the wire, containing the connection number in the protocol field;

0x80 a compressed-TCP IP datagram (COMPRESSED_TCP), with that byte being the first byte of the compressed TCP/IP datagram header;

for UNCOMPRESSED_TCP, the rest of the modified IP header, and for COMPRESSED_TCP, the compressed TCP/IP datagram header;

for a total of 16 bytes; the uncompressed IP datagram follows the header.

DLT_PPP
PPP; if the first 2 bytes are 0xff and 0x03, it's PPP in HDLC-like framing, with the PPP header following those two bytes, otherwise it's PPP without framing, and the packet begins with the PPP header.

DLT_FDDI
FDDI

DLT_ATM_RFC1483
RFC 1483 LLC/SNAP-encapsulated ATM; the packet begins with an IEEE 802.2 LLC header.

DLT_RAW
raw IP; the packet begins with an IP header.

DLT_PPP_SERIAL
PPP in HDLC-like framing, as per RFC 1662, or Cisco PPP with HDLC framing, as per section 4.3.1 of RFC 1547; the first byte will be 0xFF for PPP in HDLC-like framing, and will be 0x0F or 0x8F for Cisco PPP with HDLC framing.

DLT_PPP_ETHER
PPPoE; the packet begins with a PPPoE header, as per RFC 2516.

DLT_C_HDLC
Cisco PPP with HDLC framing, as per section 4.3.1 of RFC 1547.

DLT_IEEE802_11
IEEE 802.11 wireless LAN

DLT_FRELAY
Frame Relay

DLT_LOOP
OpenBSD loopback encapsulation; the link layer header is a 4-byte field, in *network* byte order, containing a PF_ value from OpenBSD's **socket.h** for the network-layer protocol of the packet.

Note that, if a ''savefile'' is being read, those PF_ values are *not* necessarily those of the machine reading the capture file.

DLT_LINUX_SLL
Linux "cooked" capture encapsulation; the link layer header contains, in order:

a 2-byte "packet type", in network byte order, which is one of:

0 packet was sent to us by somebody else

1 packet was broadcast by somebody else

2 packet was multicast, but not broadcast, by somebody else

3 packet was sent by somebody else to somebody else

4 packet was sent by us

a 2-byte field, in network byte order, containing a Linux ARPHRD_ value for the link layer device type;

a 2-byte field, in network byte order, containing the length of the link layer address of the sender of the packet (which could be 0);

an 8-byte field containing that number of bytes of the link layer header (if there are more than 8 bytes, only the first 8 are present);

a 2-byte field containing an Ethernet protocol type, in network byte order, or containing 1 for Novell 802.3 frames without an 802.2 LLC header or 4 for frames beginning with an 802.2 LLC header.

DLT_LTALK
Apple LocalTalk; the packet begins with an AppleTalk LLAP header.

DLT_PFLOG
OpenBSD pflog; the link layer header contains, in order:

a 1-byte header length, in host byte order;

a 4-byte PF_ value, in host byte order;

a 2-byte action code, in network byte order, which is one of:

0 passed

1 dropped

2 scrubbed

a 2-byte reason code, in network byte order, which is one of:

0 match

1 bad offset

2 fragment

3 short

4 normalize

5 memory

a 16-character interface name;

a 16-character ruleset name (only meaningful if subrule is set);

a 4-byte rule number, in network byte order;

a 4-byte subrule number, in network byte order;

a 1-byte direction, in network byte order, which is one of:

0 incoming or outgoing

1 incoming

2 outgoing

DLT_PRISM_HEADER
Prism monitor mode information followed by an 802.11 header.

DLT_IP_OVER_FC
RFC 2625 IP-over-Fibre Channel, with the link-layer header being the Network_Header as described in that RFC.

DLT_SUNATM
SunATM devices; the link layer header contains, in order:

a 1-byte flag field, containing a direction flag in the uppermost bit, which is set for packets transmitted by the machine and clear for packets received by the machine, and a 4-byte traffic type in the low-order 4 bits, which is one of:

0 raw traffic

1 LANE traffic

2 LLC-encapsulated traffic

3 MARS traffic

4 IFMP traffic

5 ILMI traffic

6 Q.2931 traffic

a 1-byte VPI value;

a 2-byte VCI field, in network byte order.

DLT_IEEE802_11_RADIO
link-layer information followed by an 802.11 header - see http://www.shaftnet.org/~pizza/software/capturefrm.txt for a description of the link-layer information.

DLT_ARCNET_LINUX
ARCNET, with no exception frames, reassembled packets rather than raw frames, and an extra 16-bit offset field between the destination host and type bytes.

DLT_LINUX_IRDA
Linux-IrDA packets, with a **DLT_LINUX_SLL** header followed by the IrLAP header.

pcap_list_datalinks() is used to get a list of the supported data link types of the interface associated with the pcap descriptor. **pcap_list_datalinks()** allocates an array to hold the list and sets **dlt_buf*. The caller is responsible for freeing the array. **−1** is returned on failure; otherwise, the number of data link types in the array is returned.

pcap_set_datalink() is used to set the current data link type of the pcap descriptor to the type specified by *dlt*. **−1** is returned on failure.

pcap_datalink_name_to_val() translates a data link type name, which is a **DLT_** name with the **DLT_** removed, to the corresponding data link type value. The translation is case-insensitive. **−1** is returned on failure.

pcap_datalink_val_to_name() translates a data link type value to the corresponding data link type name. NULL is returned on failure.

pcap_datalink_val_to_description() translates a data link type value to a short description of that data link type. NULL is returned on failure.

pcap_snapshot() returns the snapshot length specified when **pcap_open_live()** was called.

pcap_is_swapped() returns true if the current ‘‘savefile’’ uses a different byte order than the current system.

pcap_major_version() returns the major number of the file format of the savefile; **pcap_minor_version()** returns the minor number of the file format of the savefile. The version number is stored in the header of the savefile.

pcap_file() returns the standard I/O stream of the ‘‘savefile,’’ if a ‘‘savefile’’ was opened with **pcap_open_offline()**, or NULL, if a network device was opened with **pcap_open_live()**.

pcap_stats() returns 0 and fills in a **pcap_stat** struct. The values represent packet statistics from the start of the run to the time of the call. If there is an error or the underlying packet capture doesn’t support packet

statistics, −1 is returned and the error text can be obtained with **pcap_perror()** or **pcap_geterr()**. **pcap_stats()** is supported only on live captures, not on ‘‘savefiles’’; no statistics are stored in ‘‘savefiles’’, so no statistics are available when reading from a ‘‘savefile’’.

pcap_fileno() returns the file descriptor number from which captured packets are read, if a network device was opened with **pcap_open_live()**, or −1, if a ‘‘savefile’’ was opened with **pcap_open_offline()**.

pcap_get_selectable_fd() returns, on UNIX, a file descriptor number for a file descriptor on which one can do a **select()** or **poll()** to wait for it to be possible to read packets without blocking, if such a descriptor exists, or −1, if no such descriptor exists. Some network devices opened with **pcap_open_live()** do not support **select()** or **poll()** (for example, regular network devices on FreeBSD 4.3 and 4.4, and Endace DAG devices), so −1 is returned for those devices.

Note that on most versions of most BSDs (including Mac OS X) **select()** and **poll()** do not work correctly on BPF devices; **pcap_get_selectable_fd()** will return a file descriptor on most of those versions (the exceptions being FreeBSD 4.3 and 4.4), a simple **select()** or **poll()** will not return even after a timeout specified in **pcap_open_live()** expires. To work around this, an application that uses **select()** or **poll()** to wait for packets to arrive must put the **pcap_t** in non-blocking mode, and must arrange that the **select()** or **poll()** have a timeout less than or equal to the timeout specified in **pcap_open_live()**, and must try to read packets after that timeout expires, regardless of whether **select()** or **poll()** indicated that the file descriptor for the **pcap_t** is ready to be read or not. (That workaround will not work in FreeBSD 4.3 and later; however, in FreeBSD 4.6 and later, **select()** and **poll()** work correctly on BPF devices, so the workaround isn’t necessary, although it does no harm.)

pcap_get_selectable_fd() is not available on Windows.

pcap_perror() prints the text of the last pcap library error on **stderr**, prefixed by *prefix*.

pcap_geterr() returns the error text pertaining to the last pcap library error. **NOTE**: the pointer it returns will no longer point to a valid error message string after the **pcap_t** passed to it is closed; you must use or copy the string before closing the **pcap_t**.

pcap_strerror() is provided in case **strerror**(1) isn’t available.

pcap_lib_version() returns a pointer to a string giving information about the version of the libpcap library being used; note that it contains more information than just a version number.

pcap_close() closes the files associated with *p* and deallocates resources.

pcap_dump_file() returns the standard I/O stream of the ‘‘savefile’’ opened by **pcap_dump_open().**

pcap_dump_flush() flushes the output buffer to the ‘‘savefile,’’ so that any packets written with **pcap_dump()** but not yet written to the ‘‘savefile’’ will be written. **−1** is returned on error, 0 on success.

pcap_dump_ftell() returns the current file position for the ‘‘savefile’’, representing the number of bytes written by **pcap_dump_open()** and **pcap_dump()**. **−1** is returned on error.

pcap_dump_close() closes the ‘‘savefile.’’

SEE ALSO

tcpdump(1), tcpslice(1)

AUTHORS

The original authors are:

Van Jacobson, Craig Leres and Steven McCanne, all of the Lawrence Berkeley National Laboratory, University of California, Berkeley, CA.

The current version is available from "The Tcpdump Group"’s Web site at

http://www.tcpdump.org/

BUGS

Please send problems, bugs, questions, desirable enhancements, etc. to:

tcpdump-workers@tcpdump.org

Please send source code contributions, etc. to:

patches@tcpdump.org

NAME

pci — library interface for PCI bus access

LIBRARY

PCI Bus Access Library (libpci, −lpci)

SYNOPSIS

```
#include <pci.h>

int
pcibus_conf_read(int pcifd, u_int bus, u_int dev, u_int func, u_int reg,
    pcireg_t *valp);

int
pcibus_conf_write(int pcifd, u_int bus, u_int dev, u_int func, u_int reg,
    pcireg_t val);

int
pcidev_conf_read(int devfd, u_int reg, pcireg_t *valp);

int
pcidev_conf_write(int devfd, u_int reg, pcireg_t val);

char *
pci_findvendor(pcireg_t id_reg);

void
pci_devinfo(pcireg_t id_reg, pcireg_t class_reg, char *devinfo, size_t len);

void
pci_conf_print(int pcifd, u_int bus, u_int dev, u_int func);
```

DESCRIPTION

The **pci** library provides support for accessing the PCI bus by user programs.

These functions are available in the **libpci** library. Programs should be linked with **-lpci**.

CONFIGURATION SPACE FUNCTIONS

The following functions are used to access PCI configuration space:

pcibus_conf_read()
: Access the PCI configuration register *reg* on the device located at *bus*, *dev*, *func*, and place the result in **valp*. *pcifd* must be an open file descriptor to a PCI bus within the target PCI domain.

pcibus_conf_write()
: Write the value specified by *val* into the PCI configuration register *rag* on the device located at *bus*, *dev*, *func*. *pcifd* must be an open file descriptor to a PCI bus within the target PCI domain.

pcidev_conf_read()
: Access the PCI configuration register *reg* on the device associated with the open file descriptor *devfd* and place the result in **valp*.

pcidev_conf_write()
: Write the value specified by *val* into the PCI configuration register *reg* on the device associated with the open file descriptor *devfd*.

MISCELLANEOUS FUNCTIONS

The following miscellaneous functions are available:

pci_findvendor()
Return an ASCII description of the PCI vendor in the PCI ID register *id_reg*.

pci_devinfo()
Return an ASCII description of the PCI vendor, PCI product, and PCI class specified by the PCI ID register *id_reg* and PCI class ID register *class_reg*. The description is placed into the buffer pointed to by *devinfo*; the size of that buffer is specified in *len*.

pci_conf_print()
Print the PCI configuration information for the device located at *bus*, *dev*, *func*. *pcifd* must be an open file descriptor to a PCI bus within the target PCI domain.

RETURN VALUES

The **pcibus_conf_read**(), **pcibus_conf_write**(), **pcidev_conf_read**(), and **pcidev_conf_write**() functions return 0 on success and -1 on failure.

The **pci_findvendor**() function returns NULL if the PCI vendor description cannot be found.

SEE ALSO

pci(4)

HISTORY

The **pcibus_conf_read**(), **pcibus_conf_write**(), **pcidev_conf_read**(), **pcidev_conf_write**(), **pci_findvendor**(), **pci_devinfo**(), and **pci_conf_print**() functions first appeared in NetBSD 1.6.

NAME

pidfile — write a daemon pid file

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
pidfile(const char *basename);
```

DESCRIPTION

pidfile() writes a file containing the process ID of the program to the /var/run directory. The file name has the form /var/run/basename.pid. If the *basename* argument is NULL, **pidfile** will determine the program name and use that instead.

The pid file can be used as a quick reference if the process needs to be sent a signal. When the program exits, the pid file will be removed automatically, unless the program receives a fatal signal.

Note that only the first invocation of **pidfile** causes a pid file to be written; subsequent invocations have no effect unless a new *basename* is supplied. If called with a new *basename*, **pidfile**() will remove the old pid file and write the new one.

RETURN VALUES

pidfile() returns 0 on success and -1 on failure.

SEE ALSO

atexit(3)

HISTORY

The **pidfile** function call appeared in NetBSD 1.5.

BUGS

pidfile() uses atexit(3) to ensure the pidfile is unlinked at program exit. However, programs that use the _exit(2) function (for example, in signal handlers) will not trigger this behaviour.

NAME

pidlock, **ttylock**, **ttyunlock** — locks based on files containing PIDs

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
pidlock(const char *lockfile, int flags, pid_t *locker, const char *info);

int
ttylock(const char *tty, int flags, pid_t *locker);

int
ttyunlock(const char *tty);
```

DESCRIPTION

The **pidlock**() **ttylock**(), and **ttyunlock**() functions attempt to create a lockfile for an arbitrary resource that only one program may hold at a time. (In the case of **ttylock**(), this is access to a tty device.) If the function succeeds in creating the lockfile, it will succeed for no other program calling it with the same lockfile until the original calling program has removed the lockfile or exited. The **ttyunlock**() function will remove the lockfile created by **ttylock**().

These functions use the method of creating a lockfile traditionally used by UUCP software. This is described as follows in the documentation for Taylor UUCP:

> The lock file normally contains the process ID of the locking process. This makes it easy to determine whether a lock is still valid. The algorithm is to create a temporary file and then link it to the name that must be locked. If the link fails because a file with that name already exists, the existing file is read to get the process ID. If the process still exists, the lock attempt fails. Otherwise the lock file is deleted and the locking algorithm is retried.

The PID is stored in ASCII format, with leading spaces to pad it out to ten characters, and a terminating newline. This implementation has been extended to put the hostname on the second line of the file, terminated with a newline, and optionally an arbitrary comment on the third line of the file, also terminated with a newline. If a comment is given, but `PIDLOCK_NONBLOCK` is not, a blank line will be written as the second line of the file.

The **pidlock**() function will attempt to create the file *lockfile* and put the current process's pid in it. The **ttylock**() function will do the same, but should be passed only the base name (with no leading directory prefix) of the *tty* to be locked; it will test that the tty exists in `/dev` and is a character device, and then create the file in the `/var/spool/lock` directory and prefix the filename with `LCK..`. Use the **ttyunlock**() function to remove this lock.

The following flags may be passed in `flags`:

`PIDLOCK_NONBLOCK`
: The function should return immediately when a lock is held by another active process. Otherwise the function will wait (forever, if necessary) for the lock to be freed.

`PIDLOCK_USEHOSTNAME`
: The hostname should be compared against the hostname in the second line of the file (if present), and if they differ, no attempt at checking for a living process holding the lock will be made, and the lockfile will never be deleted. (The process is assumed to

be alive.) This is used for locking on NFS or other remote filesystems. (The function will never create a lock if `PIDLOCK_USEHOSTNAME` is specified and no hostname is present.)

If `locker` is non-null, it will contain the PID of the locking process, if there is one, on return.

If `info` is non-null and the lock succeeds, the string it points to will be written as the third line of the lock file.

RETURN VALUES

Zero is returned if the operation was successful; on an error a -1 is returned and a standard error code is left in the global location *errno*.

ERRORS

In addition to the errors that are returned from `stat`(2), `open`(2), `read`(2), `write`(2), and `link`(2), **`pidlock`**() or **`ttylock`**() can set *errno* to the following values on failure:

`[EWOULDBLOCK]` Another running process has a lock and the `PIDLOCK_NONBLOCK` flag was specified.

`[EFTYPE]` The *`tty`* specified in **`ttylock`**() is not a character special device.

HISTORY

The **`pidlock`**() and **`ttylock`**() functions appeared in NetBSD 1.3.

AUTHORS

Curt Sampson ⟨cjs@NetBSD.org⟩.

BUGS

The lockfile format breaks if a pid is longer than ten digits when printed in decimal form.

The PID returned will be the pid of the locker on the remote machine if `PIDLOCK_USEHOSTNAME` is specified, but there is no indication that this is not on the local machine.

NAME

pmc_configure_counter, **pmc_start_counter**, **pmc_stop_counter**, **pmc_get_num_counters**, **pmc_get_counter_class**, **pmc_get_counter_type**, **pmc_get_counter_value**, **pmc_get_accumulated_counter_value**, **pmc_get_counter_class_name**, **pmc_get_counter_type_name**, **pmc_get_counter_event_name**, **pmc_get_counter_event_list** — performance counter interface library

LIBRARY

Performance Counters Library (libpmc, −lpmc)

SYNOPSIS

```
#include <pmc.h>

int
pmc_configure_counter(int ctr, const char *evname, pmc_ctr_t reset_val,
    uint32_t flags);

int
pmc_start_counter(int ctr);

int
pmc_stop_counter(int ctr);

int
pmc_get_num_counters(void);

int
pmc_get_counter_class(void);

int
pmc_get_counter_type(int ctr, int *typep);

int
pmc_get_counter_value(int ctr, uint64_t *valp);

int
pmc_get_accumulated_counter_value(int ctr, uint64_t *valp);

const char *
pmc_get_counter_class_name(int class);

const char *
pmc_get_counter_type_name(int type);

const char *
pmc_get_counter_event_name(pmc_evid_t event);

const struct pmc_event *
pmc_get_counter_event_list(void);
```

DESCRIPTION

The **pmc** library is an interface to performance monitoring counters available on some CPUs.

The **pmc** library can count events on the following CPU families. Each second-level entry describes a performance counter class. A given class may apply to multiple individual CPU models. Each class has one or more counter types. A CPU may have more than one counter of a given type. Refer to the corresponding processor programmer's manual for more information about individual events.

- ARM
  - Intel i80200 (PMC_CLASS_I80200)

    There are two types of counters available in this class:

    PMC_TYPE_I80200_CCNT cycle counter

    PMC_TYPE_I80200_PMCx performance counter

    The following events may be counted by a counter of type PMC_TYPE_I80200_CCNT:

    clock
    clock-div-64

    The following events may be counted by a counter of type PMC_TYPE_I80200_PMCx:

    insfetch-miss
    insfetch-stall
    datadep-stall
    itlb-miss
    dtlb-miss
    branch-taken
    branch-mispredicted
    instruction-executed
    dcachebufffull-stall-time
    dcachebufffull-stall-count
    dcache-access
    dcache-miss
    dcache-writeback
    swchange-pc
    bcu-mem-request
    bcu-queue-full
    bcu-queue-drain
    bcu-ecc-no-elog
    bcu-1bit-error
    narrow-ecc-caused-rmw
- i386
  - Intel i586 (PMC_CLASS_I586)

    There are two types of counters available in this class:

    PMC_TYPE_I586_TSC cycle counter

    PMC_TYPE_I586_PMCx performance counter

    The following events may be counted by a counter of type PMC_TYPE_I586_PMCx:

    tlb-data-miss
    tlb-ins-miss
    l1cache-ins-miss
    l1cache-data-miss
    l1cache-data-miss-read
    l1cache-data-miss-write
    l1cache-writeback

l1cache-writeback-hit
l2cache-data-snoop
l2cache-data-snoop-hit
mem-read
mem-write
mem-access
mem-access-both-pipes
mem-bank-conflicts
mem-misalign-ref
mem-uncached-read
seg-load-any
branch
branch-btb-hit
branch-taken
ins-read
ins-pipeline-flush
ins-executed
ins-executed-vpipe
ins-stall-agi
ins-stall-write
ins-stall-data
ins-stall-writeline
bus-utilization
bus-locked
bus-io-cycle
fpu-flops
int-hw
break-match0
break-match1
break-match2
break-match3

- Intel i686 (PMC_CLASS_I686)

  There are two types of counters available in this class:

  PMC_TYPE_I686_TSC cycle counter

  PMC_TYPE_I686_PMCx performance counter

  The following events may be counted by a counter of type PMC_TYPE_I686_PMCx:

  mem-refs
  l1cache-lines
  l1cache-mlines
  l1cache-mlines-evict
  l1cache-miss-wait
  ins-fetch
  ins-fetch-misses
  itlb-misses
  insfetch-mem-stall
  insfetch-decode-stall

l2cache-insfetch
l2cache-data-loads
l2cache-data-stores
l2cache-lines
l2cache-lines-evict
l2cache-mlines
l2cache-mlines-evict
l2cache-reqs
l2cache-addr-strobes
l2cache-data-busy
l2cache-data-busy-read
bus-drdy-clocks-self
bus-drdy-clocks-any
bus-lock-clocks-self
bus-lock-clocks-any
bus-req-outstanding-self
bus-req-outstanding-any
bus-burst-reads-self
bus-burst-reads-any
bus-read-for-ownership-self
bus-read-for-ownership-any
bus-write-back-self
bus-write-back-any
bus-ins-fetches-self
bus-ins-fetches-any
bus-invalidates-self
bus-invalidates-any
bus-partial-writes-self
bus-partial-writes-any
bus-partial-trans-self
bus-partial-trans-any
bus-io-trans-self
bus-io-trans-any
bus-deferred-trans-self
bus-deferred-trans-any
bus-burst-trans-self
bus-burst-trans-any
bus-total-trans-self
bus-total-trans-any
bus-mem-trans-self
bus-mem-trans-any
bus-recv-cycles
bus-bnr-cycles
bus-hit-cycles
bus-hitm-cycles
bus-snoop-stall
fpu-flops
fpu-comp-ops
fpu-except-assist

fpu-mul
fpu-div
fpu-div-busy
mem-sb-blocks
mem-sb-drains
mem-misalign-ref
ins-pref-dispatch-nta
ins-pref-dispatch-t1
ins-pref-dispatch-t2
ins-pref-dispatch-weak
ins-pref-miss-nta
ins-pref-miss-t1
ins-pref-miss-t2
ins-pref-miss-weak
ins-retired
uops-retired
ins-decoded
ins-stream-retired-packed-scalar
ins-stream-retired-scalar
ins-stream-comp-retired-packed-scalar
ins-stream-comp-retired-scalar
int-hw
int-cycles-masked
int-cycles-masked-pending
branch-retired
branch-miss-retired
branch-taken-retired
branch-taken-mispred-retired
branch-decoded
branch-btb-miss
branch-bogus
branch-baclear
stall-resource
stall-partial
seg-loads
unhalted-cycles
mmx-exec
mmx-sat-exec
mmx-uops-exec
mmx-exec-packed-mul
mmx-exec-packed-shift
mmx-exec-pack-ops
mmx-exec-unpack-ops
mmx-exec-packed-logical
mmx-exec-packed-arith
mmx-trans-mmx-float
mmx-trans-float-mmx
mmx-assist
mmx-retire

seg-rename-stalls-es
seg-rename-stalls-ds
seg-rename-stalls-fs
seg-rename-stalls-gs
seg-rename-stalls-all
seg-rename-es
seg-rename-ds
seg-rename-fs
seg-rename-gs
seg-rename-all
seg-rename-retire

- AMD Athlon / K7 (PMC_CLASS_K7)

  There are two types of counters available in this class:

  PMC_TYPE_K7_TSC cycle counter

  PMC_TYPE_K7_PMCx performance counter

  The following events may be counted by a counter of type PMC_TYPE_K7_PMCx:

  seg-load-all
  seg-load-es
  seg-load-cs
  seg-load-ss
  seg-load-ds
  seg-load-fs
  seg-load-gs
  seg-load-hs
  seg-load-stall
  l1cache-access
  l1cache-miss
  l1cache-refill
  l1cache-refill-invalid
  l1cache-refill-shared
  l1cache-refill-exclusive
  l1cache-refill-owner
  l1cache-refill-modified
  l1cache-load
  l1cache-load-invalid
  l1cache-load-shared
  l1cache-load-exclusive
  l1cache-load-owner
  l1cache-load-modified
  l1cache-writeback
  l1cache-writeback-invalid
  l1cache-writeback-shared
  l1cache-writeback-exclusive
  l1cache-writeback-owner
  l1cache-writeback-modified
  l2cache-access

l2cache-tag-read
l2cache-tag-write
l2cache-inst-read
l2cache-inst-load
l2cache-data-store
l2cache-data-loadmem
l2cache-data-write
l2cache-data-move
l2cache-access-busy
l2cache-hit
l2cache-miss
mem-misalign-ref
mem-access
mem-access-uc
mem-access-wc
mem-access-wt
mem-access-wp
mem-access-wb
ins-fetch
ins-fetch-miss
ins-refill-l2
ins-refill-mem
ins-fetch-stall
ins-retired
ins-empty
itlb-miss-l1
itlb-miss-l2
ops-retired
branch-retired
branch-miss-retired
branch-taken-retired
branch-taken-miss-retired
branch-far-retired
branch-resync-retired
branch-near-retired
branch-near-miss-retired
branch-indirect-miss-retired
int-hw
int-cycles-masked
int-cycles-masked-pending
break-match0
break-match1
break-match2
break-match3

The **pmc** library maintains a mapping between event names and the event selector used by the CPU's performance monitoring hardware. The mapping is described by the following structure:

```
struct pmc_event {
        const char *name;
        pmc_evid_t val;
};
```

The **pmc_configure_counter**() function configures the counter *ctr* to count the event *evname*. The initial value of the counter will be set to *reset_val*, and this value will be loaded back into the counter each time it overflows. There are currently no flags defined for the *flags* argument.

The **pmc_start_counter**() function enables counting on counter *ctr*.

The **pmc_stop_counter**() function disables counting on counter *ctr*.

The **pmc_get_num_counters**() function returns the number of counters present in the CPU.

The **pmc_get_counter_class**() function returns the counter class of the CPU.

The **pmc_get_counter_type**() function places the counter type of counter *ctr* into ∗*typep*.

The **pmc_get_counter_value**() function places the total number of events counted by counter *ctr* into ∗*valp*.

The **pmc_get_accumulated_counter_value**() function places the total number of events counted for the current process and all of its children by counter *ctr* into ∗*valp*.

The **pmc_get_counter_class_name**() function returns the name of the counter class *classval*.

The **pmc_get_counter_type_name**() function returns the name of the counter type *type*.

The **pmc_get_counter_event_name**() function returns the name of the event *event* for the current CPU's performance counter class.

The **pmc_get_counter_event_list**() function returns an array of *pmc_event* structures, listing the supported event types for the CPU. The array is terminated by and entry who's *name* member is NULL.

RETURN VALUES

The **pmc_configure_counter**(), **pmc_start_counter**(), **pmc_stop_counter**(), **pmc_get_counter_type**(), **pmc_get_counter_value**(), and **pmc_get_accumulated_counter_value**() functions return 0 to indicate success and −1 to indicate failure, in which case errno(2) will be set to indicate the mode of failure.

The **pmc_get_counter_class_name**(), **pmc_get_counter_type_name**(), **pmc_get_counter_event_name**(), and **pmc_get_counter_event_list**() functions return NULL and set errno(2) to indicate failure.

SEE ALSO

pmc(1), pmc_control(2), pmc_get_info(2), pmc(9)

HISTORY

The **pmc** library first appeared in NetBSD 2.0.

AUTHORS

The **pmc** library was written by Jason R. Thorpe ⟨thorpej@wasabisystems.com⟩ and contributed by Wasabi Systems, Inc.

NAME

popen, **pclose** — process I/O

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

FILE *
popen(const char *command, const char *type);

int
pclose(FILE *stream);
```

DESCRIPTION

The **popen**() function "opens" a process by creating an IPC connection, forking, and invoking the shell. Historically, **popen** was implemented with a unidirectional pipe; hence many implementations of **popen** only allow the *type* argument to specify reading or writing, not both. Since **popen** is now implemented using sockets, the *type* may request a bidirectional data flow. The *type* argument is a pointer to a null-terminated string which must be 'r' for reading, 'w' for writing, or 'r+' for reading and writing.

The *command* argument is a pointer to a null-terminated string containing a shell command line. This command is passed to /bin/sh using the **-c** flag; interpretation, if any, is performed by the shell.

The return value from **popen**() is a normal standard I/O stream in all respects save that it must be closed with **pclose**() rather than **fclose**(). Writing to such a stream writes to the standard input of the command; the command's standard output is the same as that of the process that called **popen**(), unless this is altered by the command itself. Conversely, reading from a "popened" stream reads the command's standard output, and the command's standard input is the same as that of the process that called **popen**().

Note that output **popen**() streams are fully buffered by default.

The **pclose**() function waits for the associated process to terminate and returns the exit status of the command as returned by **wait4**().

RETURN VALUES

The **popen**() function returns NULL if the fork(2), pipe(2), or socketpair(2) calls fail, or if it cannot allocate memory.

The **pclose**() function returns −1 if *stream* is not associated with a "popened" command, if *stream* has already been "pclosed", or if wait4(2) returns an error.

ERRORS

The **popen**() function does not reliably set *errno*.

SEE ALSO

sh(1), fork(2), pipe(2), socketpair(2), wait4(2), fclose(3), fflush(3), fopen(3), shquote(3), stdio(3), system(3)

STANDARDS

The **popen**() and **pclose**() functions conform to IEEE Std 1003.2-1992 ("POSIX.2").

HISTORY

A **popen**() and a **pclose**() function appeared in Version 7 AT&T UNIX.

BUGS

Since the standard input of a command opened for reading shares its seek offset with the process that called **popen**(), if the original process has done a buffered read, the command's input position may not be as expected. Similarly, the output from a command opened for writing may become intermingled with that of the original process. The latter can be avoided by calling `fflush`(3) before **popen**().

Failure to execute the shell is indistinguishable from the shell's failure to execute command, or an immediate exit of the command. The only hint is an exit status of 127.

The **popen**() argument always calls `sh`(1), never calls `csh`(1).

NAME

posix_memalign — aligned memory allocation

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
posix_memalign(void **ptr, size_t alignment, size_t size);
```

DESCRIPTION

The **posix_memalign**() function allocates *size* bytes of memory such that the allocation's base address is an even multiple of *alignment*, and returns the allocation in the value pointed to by *ptr*.

The requested *alignment* must be a power of 2 at least as large as **sizeof**(*void* *).

Memory that is allocated via **posix_memalign**() can be used as an argument in subsequent calls to realloc(3) and free(3).

RETURN VALUES

The **posix_memalign**() function returns the value 0 if successful; otherwise it returns an error value.

ERRORS

The **posix_memalign**() function will fail if:

[EINVAL] The *alignment* parameter is not a power of 2 at least as large as **sizeof**(*void* *).

[ENOMEM] Memory allocation error.

SEE ALSO

free(3), malloc(3), realloc(3), valloc(3)

STANDARDS

The **posix_memalign**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

posix_openpt — open a pseudo-terminal device

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>
#include <fcntl.h>

int
posix_openpt(int oflag);
```

DESCRIPTION

posix_openpt() Searches for an unused master pseudo-terminal device, opens it, and returns a file descriptor associated the now used pseudo-terminal device. The *oflag* argument has the same meaning as in the open(2) call.

RETURN VALUES

If successful, **posix_openpt**() returns a non-negative integer, which corresponds to a file descriptor pointing to the master pseudo-terminal device. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

ioctl(2), open(2), grantpt(3), ptsname(3), unlockpt(3)

RATIONALE

The standards committee did not want to directly expose the cloning device, thus decided to wrap the functionality in this function. The equivalent code would be:

```
int
posix_openpt(int oflag) {
        return open("/dev/ptmx", oflag);
}
```

STANDARDS

The **posix_openpt**() function conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

printf, **fprintf**, **sprintf**, **snprintf**, **asprintf**, **vprintf**, **vfprintf**, **vsprintf**, **vsnprintf**, **vasprintf** — formatted output conversion

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

int
printf(const char * restrict format, ...);

int
fprintf(FILE * restrict stream, const char * restrict format, ...);

int
sprintf(char * restrict str, const char * restrict format, ...);

int
snprintf(char * restrict str, size_t size, const char * restrict format,
    ...);

int
asprintf(char ** restrict ret, const char * restrict format, ...);

#include <stdarg.h>

int
vprintf(const char * restrict format, va_list ap);

int
vfprintf(FILE * restrict stream, const char * restrict format, va_list ap);

int
vsprintf(char * restrict str, const char * restrict format, va_list ap);

int
vsnprintf(char * restrict str, size_t size, const char * restrict format,
    va_list ap);

int
vasprintf(char ** restrict ret, const char * restrict format, va_list ap);
```

DESCRIPTION

The **printf**() family of functions produces output according to a *format* as described below. The **printf**() and **vprintf**() functions write output to *stdout*, the standard output stream; **fprintf**() and **vfprintf**() write output to the given output *stream*; **sprintf**(), **snprintf**(), **vsprintf**(), and **vsnprintf**() write to the character string *str*; and **asprintf**() and **vasprintf**() write to a dynamically allocated string that is stored in *ret*.

These functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of `stdarg`(3)) are converted for output.

These functions return the number of characters printed (not including the trailing '`\0`' used to end output to strings). If an output error was encountered, these functions shall return a negative value.

asprintf() and **vasprintf**() return a pointer to a buffer sufficiently large to hold the string in the *ret* argument. This pointer should be passed to free(3) to release the allocated storage when it is no longer needed. If sufficient space cannot be allocated, these functions will return −1 and set *ret* to be a NULL pointer. Please note that these functions are not standardized, and not all implementations can be assumed to set the *ret* argument to NULL on error. It is more portable to check for a return value of −1 instead.

snprintf() and **vsnprintf**() will write at most *size*−1 of the characters printed into the output string (the *size*'th character then gets the terminating '\0'); if the return value is greater than or equal to the *size* argument, the string was too short and some of the printed characters were discarded. If *size* is zero, nothing is written and *str* may be a NULL pointer.

sprintf() and **vsprintf**() effectively assume an infinite *size*.

The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**. The arguments must correspond properly (after type promotion) with the conversion specifier. After the **%**, the following appear in sequence:

- An optional field, consisting of a decimal digit string followed by a **$**, specifying the next argument to access. If this field is not provided, the argument following the last argument accessed will be used. Arguments are numbered starting at **1**. If unaccessed arguments in the format string are interspersed with ones that are accessed the results will be indeterminate.
- Zero or more of the following flags:

  '**#**' The value should be converted to an "alternate form". For **c**, **d**, **i**, **n**, **p**, **s**, and **u** conversions, this option has no effect. For **o** conversions, the precision of the number is increased to force the first character of the output string to a zero (except if a zero value is printed with an explicit precision of zero). For **x** and **X** conversions, a non-zero result has the string '0x' (or '0X' for **X** conversions) prepended to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For **g** and **G** conversions, trailing zeros are not removed from the result as they would otherwise be.

  '**0**' (zero) Zero padding. For all conversions except **n**, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (**d**, **i**, **o**, **u**, **i**, **x**, and **X**), the **0** flag is ignored.

  '**−**' A negative field width flag; the converted value is to be left adjusted on the field boundary. Except for **n** conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A '**−**' overrides a '**0**' if both are given.

  ' ' (space) A blank should be left before a positive number produced by a signed conversion (**a**, **A** **d**, **e**, **E**, **f**, **F**, **g**, **G**, or **i**).

  '**+**' A sign must always be placed before a number produced by a signed conversion. A '**+**' overrides a space if both are used.

  ''' Decimal conversions (**d**, **u**, or **i**) or the integral portion of a floating point conversion (**f** or **F**) should be grouped and separated by thousands using the non-monetary separator returned by localeconv(3).

- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given) to fill out the field width.

- An optional precision, in the form of a period '**.**' followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear for **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for **g** and **G** conversions, or the maximum number of characters to be printed from a string for **s** conversions.

- An optional length modifier, that specifies the size of the argument. The following length modifiers are valid for the **d**, **i**, **n**, **o**, **u**, **x**, or **X** conversion:

Modifier	**d**, **i**	**o**, **u**, **x**, **X**	**n**
hh	*signed char*	*unsigned char*	*signed char* *
h	*short*	*unsigned short*	*short* *
l (ell)	*long*	*unsigned long*	*long* *
ll (ell ell)	*long long*	*unsigned long long*	*long long* *
j	*intmax_t*	*uintmax_t*	*intmax_t* *
t	*ptrdiff_t*	(see note)	*ptrdiff_t* *
z	(see note)	*size_t*	(see note)
q *(deprecated)*	*quad_t*	*u_quad_t*	*quad_t* *

  Note: the **t** modifier, when applied to a **o**, **u**, **x**, or **X** conversion, indicates that the argument is of an unsigned type equivalent in size to a *ptrdiff_t*. The **z** modifier, when applied to a **d** or **i** conversion, indicates that the argument is of a signed type equivalent in size to a *size_t*. Similarly, when applied to an **n** conversion, it indicates that the argument is a pointer to a signed type equivalent in size to a *size_t*.

  The following length modifier is valid for the **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion:

Modifier	**a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**
l (ell)	*double* (ignored, same behavior as without it)
L	*long double*

  The following length modifier is valid for the **c** or **s** conversion:

Modifier	**c**	**s**
l (ell)	*wint_t*	*wchar_t* *

- A character that specifies the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk '*' or an asterisk followed by one or more decimal digits and a '$' instead of a digit string. In this case, an *int* argument supplies the field width or precision. A negative field width is treated as a left adjustment flag followed by a positive field width; a negative precision is treated as though it were missing. If a single format directive mixes positional (nn$) and non-positional arguments, the results are undefined.

The conversion specifiers and their meanings are:

diouxX The *int* (or appropriate variant) argument is converted to signed decimal (**d** and **i**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation. The letters "abcdef" are used for **x** conversions; the letters "ABCDEF" are used for **X** conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros.

DOU The *long int* argument is converted to signed decimal, unsigned octal, or unsigned decimal, as if the format had been **ld**, **lo**, or **lu** respectively. These conversion characters are deprecated, and will eventually disappear.

eE The *double* argument is rounded and converted in the style [−]*d.ddd*e±*dd* where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An **E** conversion uses the letter 'E' (rather than 'e') to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, positive and negative infinity are represented as `inf` and `-inf` respectively when using the lowercase conversion character, and `INF` and `-INF` respectively when using the uppercase conversion character. Similarly, NaN is represented as `nan` when using the lowercase conversion, and `NAN` when using the uppercase conversion.

fF The *double* argument is rounded and converted to decimal notation in the style [−]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

gG The *double* argument is converted in style **f** or **e** (or in style **F** or **E** for **G** conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style **e** is used if the exponent from its conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

aA The *double* argument is rounded and converted to hexadecimal notation in the style [−]`0x`*h.hhhp*[±]*d*, where the number of digits after the hexadecimal-point character is equal to the precision specification. If the precision is missing, it is taken as enough to represent the floating-point number exactly, and no rounding occurs. If the precision is zero, no hexadecimal-point character appears. The **p** is a literal character 'p', and the exponent consists of a positive or negative sign followed by a decimal number representing an exponent of 2. The **A** conversion uses the prefix "`0X`" (rather than "`0x`"), the letters "`ABCDEF`" (rather than "`abcdef`") to represent the hex digits, and the letter 'P' (rather than 'p') to separate the mantissa and exponent.

Note that there may be multiple valid ways to represent floating-point numbers in this hexadecimal format. For example, `0x3.24p+0`, `0x6.48p-1` and `0xc.9p-2` are all equivalent. The format chosen depends on the internal representation of the number, but the implementation guarantees that the length of the mantissa will be minimized. Zeroes are always represented with a mantissa of 0 (preceded by a '`-`' if appropriate) and an exponent of `+0`.

C Treated as **c** with the **l** (ell) modifier.

c The *int* argument is converted to an *unsigned char*, and the resulting character is written.

If the **l** (ell) modifier is used, the *wint_t* argument shall be converted to a *wchar_t*, and the (potentially multi-byte) sequence representing the single wide character is written, including any shift sequences. If a shift sequence is used, the shift state is also restored to the original state after the character.

S Treated as **s** with the **l** (ell) modifier.

s The *char* * argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating `NUL` character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating `NUL` character.

If the **l** (ell) modifier is used, the *wchar_t* * argument is expected to be a pointer to an array of wide characters (pointer to a wide string). For each wide character in the string, the (potentially multi-byte) sequence representing the wide character is written, including any shift sequences. If

any shift sequence is used, the shift state is also restored to the original state after the string. Wide characters from the array are written up to (but not including) a terminating wide NUL character; if a precision is specified, no more than the number of bytes specified are written (including shift sequences). Partial characters are never written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the number of bytes required to render the multibyte representation of the string, the array must contain a terminating wide NUL character.

p The *void* * pointer argument is printed in hexadecimal (as if by %#x or %#lx).

n The number of characters written so far is stored into the integer indicated by the *int* * (or variant) pointer argument. No argument is converted.

% A '%' is written. No argument is converted. The complete conversion specification is '%%'.

The decimal point character is defined in the program's locale (category LC_NUMERIC).

In no case does a non-existent or small field width cause truncation of a numeric field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to strings:

```
#include <stdio.h>
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
       weekday, month, day, hour, min);
```

To print π to five decimal places:

```
#include <math.h>
#include <stdio.h>
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

To allocate a 128 byte string and print into it:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
char *newfmt(const char *fmt, ...)
{
               char *p;
               va_list ap;
               if ((p = malloc(128)) == NULL)
                      return (NULL);
               va_start(ap, fmt);
               (void) vsnprintf(p, 128, fmt, ap);
               va_end(ap);
               return (p);
}
```

SECURITY CONSIDERATIONS

The **sprintf**() and **vsprintf**() functions are easily misused in a manner which enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack. Because **sprintf**() and **vsprintf**() assume an infinitely long string, callers must be careful not to overflow the actual space; this is often hard to assure. For safety, programmers should use the **snprintf**() interface instead. For example:

```
void
foo(const char *arbitrary_string, const char *and_another)
{
        char onstack[8];

#ifdef BAD
        /*
         * This first sprintf is bad behavior.  Do not use sprintf!
         */
        sprintf(onstack, "%s, %s", arbitrary_string, and_another);
#else
        /*
         * The following two lines demonstrate better use of
         * snprintf().
         */
        snprintf(onstack, sizeof(onstack), "%s, %s", arbitrary_string,
            and_another);
#endif
}
```

The **printf**() and **sprintf**() family of functions are also easily misused in a manner allowing malicious users to arbitrarily change a running program's functionality by either causing the program to print potentially sensitive data "left on the stack", or causing it to generate a memory fault or bus error by dereferencing an invalid pointer.

%n can be used to write arbitrary data to potentially carefully-selected addresses. Programmers are therefore strongly advised to never pass untrusted strings as the *format* argument, as an attacker can put format specifiers in the string to mangle your stack, leading to a possible security hole. This holds true even if the string was built using a function like **snprintf**(), as the resulting string may still contain user-supplied conversion specifiers for later interpolation by **printf**().

Always use the proper secure idiom:

```
snprintf(buffer, sizeof(buffer), "%s", string);
```

ERRORS

In addition to the errors documented for the write(2) system call, the **printf**() family of functions may fail if:

[EILSEQ]	An invalid wide character code was encountered.
[ENOMEM]	Insufficient storage space is available.

SEE ALSO

printf(1), fmtcheck(3), printf(9), scanf(3), setlocale(3), wprintf(3)

STANDARDS

Subject to the caveats noted in the **BUGS** section below, the **fprintf**(), **printf**(), **sprintf**(), **vprintf**(), **vfprintf**(), and **vsprintf**() functions conform to ANSI X3.159-1989 ("ANSI C89") and ISO/IEC 9899:1999 ("ISO C99"). With the same reservation, the **snprintf**() and **vsnprintf**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

HISTORY

The functions **snprintf**() and **vsnprintf**() first appeared in 4.4BSD. The functions **asprintf**() and **vasprintf**() are modeled on the ones that first appeared in the GNU C library.

CAVEATS

Because **sprintf**() and **vsprintf**() assume an infinitely long string, callers must be careful not to overflow the actual space; this is often impossible to assure. For safety, programmers should use the **snprintf**() and **asprintf**() family of interfaces instead. Unfortunately, the **snprintf**() interfaces are not available on older systems and the **asprintf**() interfaces are not yet portable.

It is important never to pass a string with user-supplied data as a format without using '%s'. An attacker can put format specifiers in the string to mangle your stack, leading to a possible security hole. This holds true even if you have built the string "by hand" using a function like **snprintf**(), as the resulting string may still contain user-supplied conversion specifiers for later interpolation by **printf**().

Be sure to use the proper secure idiom:

```
snprintf(buffer, sizeof(buffer), "%s", string);
```

There is no way for printf to know the size of each argument passed. If you use positional arguments you must ensure that all parameters, up to the last positionally specified parameter, are used in the format string. This allows for the format string to be parsed for this information. Failure to do this will mean your code is non-portable and liable to fail.

In this implementation, passing a NULL *char* * argument to the **%s** format specifier will output *(null)* instead of crashing. Programs that depend on this behavior are non-portable and may crash on other systems or in the future.

BUGS

The conversion formats **%D**, **%O**, and **%U** are not standard and are provided only for backward compatibility. The effect of padding the **%p** format with zeros (either by the '**0**' flag or by specifying a precision), and the benign effect (i.e. none) of the '**#**' flag on **%n** and **%p** conversions, as well as other nonsensical combinations such as **%Ld**, are not standard; such combinations should be avoided.

The **printf** family of functions do not correctly handle multibyte characters in the *format* argument.

NAME

prop_array, **prop_array_create**, **prop_array_create_with_capacity**, **prop_array_copy**, **prop_array_copy_mutable**, **prop_array_capacity**, **prop_array_count**, **prop_array_ensure_capacity**, **prop_array_iterator**, **prop_array_make_immutable**, **prop_array_mutable**, **prop_array_get**, **prop_array_set**, **prop_array_add**, **prop_array_remove**, **prop_array_externalize**, **prop_array_internalize**, **prop_array_externalize_to_file**, **prop_array_internalize_from_file**, **prop_array_equals** — array property collection object

LIBRARY

library “libprop”

SYNOPSIS

```
#include <prop/proplib.h>

prop_array_t
prop_array_create(void);

prop_array_t
prop_array_create_with_capacity(unsigned int capacity);

prop_array_t
prop_array_copy(prop_array_t array);

prop_array_t
prop_array_copy_mutable(prop_array_t array);

unsigned int
prop_array_capacity(prop_array_t array);

unsigned int
prop_array_count(prop_array_t array);

bool
prop_array_ensure_capacity(prop_array_t array, unsigned int capacity);

prop_object_iterator_t
prop_array_iterator(prop_array_t array);

void
prop_array_make_immutable(prop_array_t array);

bool
prop_array_mutable(prop_array_t array);

prop_object_t
prop_array_get(prop_array_t array, unsigned int index);

bool
prop_array_set(prop_array_t array, unsigned int index, prop_object_t obj);

bool
prop_array_add(prop_array_t array, prop_object_t obj);

void
prop_array_remove(prop_array_t array, unsigned int index);
```

*char **
prop_array_externalize(*prop_array_t array*);

prop_array_t
prop_array_internalize(*const char *xml*);

bool
prop_array_externalize_to_file(*prop_array_t array*, *const char *path*);

prop_array_t
prop_array_internalize_from_file(*const char *path*);

bool
prop_array_equals(*prop_array_t array1*, *prop_array_t array2*);

DESCRIPTION

The **prop_array** family of functions operate on the array property collection object type. An array is an ordered set; an iterated array will return objects in the same order with which they were stored.

prop_array_create(*void*)
Create an empty array. The array initially has no capacity. Returns NULL on failure.

prop_array_create_with_capacity(*unsigned int capacity*)
Create an array with the capacity to store *capacity* objects. Returns NULL on failure.

prop_array_copy(*prop_array_t array*)
Copy an array. The new array has an initial capacity equal to the number of objects stored in the array being copied. The new array contains references to the original array's objects, not copies of those objects (i.e. a shallow copy is made). If the original array is immutable, the resulting array is also immutable. Returns NULL on failure.

prop_array_copy_mutable(*prop_array_t array*)
Like **prop_array_copy**(), except the resulting array is always mutable.

prop_array_capacity(*prop_array_t array*)
Returns the total capacity of the array, including objects already stored in the array. If the supplied object isn't an array, zero is returned.

prop_array_count(*prop_array_t array*)
Returns the number of objects stored in the array. If the supplied object isn't an array, zero is returned.

prop_array_ensure_capacity(*prop_array_t array*, *unsigned int capacity*)
Ensure that the array has a total capacity of *capacity*, including objects already stored in the array. Returns true if the capacity of the array is greater or equal to *capacity* or if expansion of the array's capacity was successful and false otherwise.

prop_array_iterator(*prop_array_t array*)
Create an iterator for the array. The array is retained by the iterator. An array iterator returns the object references stored in the array. Storing to or removing from the array invalidates any active iterators for the array. Returns NULL on failure.

prop_array_make_immutable(*prop_array_t array*)
Make *array* immutable.

prop_array_mutable(*prop_array_t array*)
Returns true if the array is mutable.

prop_array_get(*prop_array_t array*, *unsigned int index*)
Return the object stored at the array index *index*. Returns NULL on failure.

prop_array_set(*prop_array_t array*, *unsigned int index*, *prop_object_t obj*)
Store a reference to the object *obj* at the array index *index*. This function is not allowed to create holes in the array; the caller must either be setting the object just beyond the existing count or replacing an already existing object reference. The object will be retained by the array. If an existing object reference is being replaced, that object will be released. Returns true if storing the object was successful and false otherwise.

prop_array_add(*prop_array_t array*, *prop_object_t obj*)
Add a reference to the object *obj* to the array, appending to the end and growing the array's capacity if necessary. The object will be retained by the array. Returns true if storing the object was successful and false otherwise.

During expansion, array's capacity is augmented by the EXPAND_STEP constant, as defined in libprop/prop_array.c file, e.g.

#define EXPAND_STEP 16

prop_array_remove(*prop_array_t array*, *unsigned int index*)
Remove the reference to the object stored at array index *index*. The object will be released and the array compacted following the removal.

prop_array_equals(*prop_array_t array1*, *prop_array_t array2*)
Returns true if the two arrays are equivalent. If at least one of the supplied objects isn't an array, false is returned. Note: Objects contained in the array are compared by value, not by reference.

prop_array_externalize(*prop_array_t array*)
Externalizes an array, returning a NUL-terminated buffer containing the XML representation of the array. The caller is responsible for freeing the returned buffer. If converting to the external representation fails for any reason, NULL is returned.

In user space, the buffer is allocated using malloc(3). In the kernel, the buffer is allocated using malloc(9) using the malloc type M_TEMP.

prop_array_internalize(*const char *xml*)
Parse the XML representation of a property list in the NUL-terminated buffer *xml* and return the corresponding array. Returns NULL if parsing fails for any reason.

prop_array_externalize_to_file(*prop_array_t array*, *const char *path*)
Externalizes an array and writes it to the file specified by *path*. The file is saved with the mode 0666 as modified by the process's file creation mask (see umask(3)) and is written atomically. Returns false if externalizing or writing the array fails for any reason.

prop_array_internalize_from_file(*const char *path*)
Reads the XML property list contained in the file specified by *path*, internalizes it, and returns the corresponding array. Returns NULL on failure.

SEE ALSO

prop_bool(3), prop_data(3), prop_dictionary(3), prop_number(3), prop_object(3), prop_string(3), proplib(3)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

NAME

prop_array_util, **prop_array_get_bool**, **prop_array_set_bool**, **prop_array_get_int8**, **prop_array_get_uint8**, **prop_array_set_int8**, **prop_array_set_uint8**, **prop_array_get_int16**, **prop_array_get_uint16**, **prop_array_set_int16**, **prop_array_set_uint16**, **prop_array_get_int32**, **prop_array_get_uint32**, **prop_array_set_int32**, **prop_array_set_uint32**, **prop_array_get_int64**, **prop_array_get_uint64**, **prop_array_set_int64**, **prop_array_set_uint64**, **prop_array_get_cstring**, **prop_array_set_cstring**, **prop_array_get_cstring_nocopy**, **prop_array_set_cstring_nocopy**

LIBRARY

library “libprop”

SYNOPSIS

```
#include <prop/proplib.h>

bool
prop_array_get_bool(prop_array_t dict, unsigned int indx, bool *valp);

bool
prop_array_set_bool(prop_array_t dict, unsigned int indx, bool val);

bool
prop_array_get_int8(prop_array_t dict, unsigned int indx, int8_t *valp);

bool
prop_array_get_uint8(prop_array_t dict, unsigned int indx, uint8_t *valp);

bool
prop_array_set_int8(prop_array_t dict, unsigned int indx, int8_t val);

bool
prop_array_set_uint8(prop_array_t dict, unsigned int indx, uint8_t val);

bool
prop_array_get_int16(prop_array_t dict, unsigned int indx, int16_t *valp);

bool
prop_array_get_uint16(prop_array_t dict, unsigned int indx,
    uint16_t *valp);

bool
prop_array_set_int16(prop_array_t dict, unsigned int indx, int16_t val);

bool
prop_array_set_uint16(prop_array_t dict, unsigned int indx, uint16_t val);

bool
prop_array_get_int32(prop_array_t dict, unsigned int indx, int32_t *valp);

bool
prop_array_get_uint32(prop_array_t dict, unsigned int indx,
    uint32_t *valp);

bool
prop_array_set_int32(prop_array_t dict, unsigned int indx, int32_t val);
```

bool
prop_array_set_uint32(*prop_array_t dict*, *unsigned int indx*, *uint32_t val*);

bool
prop_array_get_int64(*prop_array_t dict*, *unsigned int indx*, *int64_t *valp*);

bool
prop_array_get_uint64(*prop_array_t dict*, *unsigned int indx*,
*uint64_t *valp*);

bool
prop_array_set_int64(*prop_array_t dict*, *unsigned int indx*, *int64_t val*);

bool
prop_array_set_uint64(*prop_array_t dict*, *unsigned int indx*, *uint64_t val*);

bool
prop_array_get_cstring(*prop_array_t dict*, *unsigned int indx*, *char **strp*);

bool
prop_array_set_cstring(*prop_array_t dict*, *unsigned int indx*,
*const char *str*);

bool
prop_array_get_cstring_nocopy(*prop_array_t dict*, *unsigned int indx*,
*const char **strp*);

bool
prop_array_set_cstring_nocopy(*prop_array_t dict*, *unsigned int indx*,
*const char *strp*);

DESCRIPTION

The **prop_array_util** family of functions are provided to make getting and setting values in arrays more convenient in some applications.

The getters check the type of the returned object and, in some cases, also ensure that the returned value is within the range implied by the getter's value type.

The setters handle object creation and release for the caller.

The **prop_array_get_cstring**() function returns dynamically allocated memory. See `prop_string`(3) for more information.

The **prop_array_get_cstring_nocopy**() and **prop_array_set_cstring_nocopy**() functions do not copy the string that is set or returned. See `prop_string`(3) for more information.

RETURN VALUES

The **prop_array_util** getter functions return `true` if the object exists in the array and the value is in-range, or `false` otherwise.

The **prop_array_util** setter functions return `true` if creating the object and storing it in the array is successful, or `false` otherwise.

SEE ALSO

`prop_array`(3), `prop_bool`(3), `prop_number`(3), `proplib`(3)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

NAME

prop_bool, **prop_bool_create**, **prop_bool_copy**, **prop_bool_true** — boolean value property object

LIBRARY

library “libprop”

SYNOPSIS

```
#include <prop/proplib.h>

prop_bool_t
prop_bool_create(bool val);

prop_bool_t
prop_booly_copy(prop_bool_t bool);

bool
prop_bool_true(prop_bool_t bool);
```

DESCRIPTION

The **prop_bool** family of functions operate on a boolean value property object type.

prop_bool_create(*bool val*)
: Create a boolean value object with the value *val*.

prop_bool_copy(*prop_bool_t bool*)
: Copy a boolean value object. If the supplied object isn’t a boolean, `NULL` is returned.

prop_bool_true(*prop_bool_t bool*)
: Returns the value of the boolean value object. If the supplied object isn’t a boolean, `false` is returned.

SEE ALSO

`prop_array`(3), `prop_data`(3), `prop_dictionary`(3), `prop_number`(3), `prop_object`(3), `prop_string`(3), `proplib`(3)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

NAME

prop_data, **prop_data_create_data**, **prop_data_create_data_nocopy**, **prop_data_copy**, **prop_data_size**, **prop_data_data**, **prop_data_data_nocopy**, **prop_data_equals**, **prop_data_equals_data** — opaque data value property object

LIBRARY

library “libprop”

SYNOPSIS

```
#include <prop/proplib.h>

prop_data_t
prop_data_create_data(const void *blob, size_t len);

prop_data_t
prop_data_create_data_nocopy(const void *blob, size_t len);

prop_data_t
prop_data_copy(prop_data_t data);

void *
prop_data_data(prop_data_t data);

size_t
prop_data_size(prop_data_t data);

const void *
prop_data_data_nocopy(prop_data_t data);

bool
prop_data_equals(prop_data_t dat1, prop_data_t dat2);

bool
prop_data_equals_data(prop_data_t data, const void *blob, size_t len);
```

DESCRIPTION

The **prop_data** family of functions operate on an opaque data value property object type.

prop_data_create_data(*const void *blob*, *size_t len*)
: Create a data object that contains a copy of *blob* with size *len*. Returns NULL on failure.

prop_data_create_data_nocopy(*const void *blob*, *size_t len*)
: Create a data object that contains a reference to *blob* with size *len*. Returns NULL on failure.

prop_data_copy(*prop_data_t data*)
: Copy a data object. If the data object being copied is an external data reference, then the copy also references the same external data. Returns NULL on failure.

prop_data_size(*prop_data_t data*)
: Returns the size of the data object. If the supplied object isn’t a data object, zero is returned.

prop_data_data(*prop_data_t data*)
: Returns a copy of the data object’s contents. The caller is responsible for freeing the returned buffer. If the supplied object isn’t a data object or if the data container is empty, NULL is returned.

  In user space, the buffer is allocated using malloc(3). In the kernel, the buffer is allocated using malloc(9) using the malloc type M_TEMP.

prop_data_data_nocopy(*prop_data_t data*)
Returns an immutable reference to the contents of the data object. If the supplied object isn't a data object, NULL is returned.

prop_data_equals(*prop_data_t dat1*, *prop_data_t dat2*)
Returns true if the two data objects are equivalent. If at least one of the supplied objects isn't a data object, false is returned.

prop_data_equals_data(*prop_data_t data*, *const void *blob*, *size_t len*)
Returns true if the data object's value is equivalent to *blob* with size *len*. If the supplied object isn't a data object, false is returned.

SEE ALSO

prop_array(3), prop_bool(3), prop_dictionary(3), prop_number(3), prop_object(3), prop_string(3), proplib(3)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

NAME

prop_dictionary, **prop_dictionary_create**, **prop_dictionary_create_with_capacity**, **prop_dictionary_copy**, **prop_dictionary_copy_mutable**, **prop_dictionary_count**, **prop_dictionary_ensure_capacity**, **prop_dictionary_iterator**, **prop_dictionary_all_keys**, **prop_dictionary_make_immutable**, **prop_dictionary_mutable**, **prop_dictionary_get**, **prop_dictionary_set**, **prop_dictionary_remove**, **prop_dictionary_get_keysym**, **prop_dictionary_set_keysym**, **prop_dictionary_remove_keysym**, **prop_dictionary_externalize**, **prop_dictionary_internalize**, **prop_dictionary_externalize_to_file**, **prop_dictionary_internalize_from_file**, **prop_dictionary_equals**, **prop_dictionary_keysym_cstring_nocopy**, **prop_dictionary_keysym_equals** — dictionary property collection object

LIBRARY

library “libprop”

SYNOPSIS

```
#include <prop/proplib.h>

prop_dictionary_t
prop_dictionary_create(void);

prop_dictionary_t
prop_dictionary_create_with_capacity(unsigned int capacity);

prop_dictionary_t
prop_dictionary_copy(prop_dictionary_t dict);

prop_dictionary_t
prop_dictionary_copy_mutable(prop_dictionary_t dict);

unsigned int
prop_dictionary_count(prop_dictionary_t dict);

bool
prop_dictionary_ensure_capacity(prop_dictionary_t dict,
    unsigned int capacity);

prop_object_iterator_t
prop_dictionary_iterator(prop_dictionary_t dict);

prop_array_t
prop_dictionary_all_keys(prop_dictionary_t dict);

void
prop_dictionary_make_immutable(prop_dictionary_t dict);

bool
prop_dictionary_mutable(prop_dictionary_t dict);

prop_object_t
prop_dictionary_get(prop_dictionary_t dict, const char *key);

bool
prop_dictionary_set(prop_dictionary_t dict, const char *key,
    prop_object_t obj);
```

void
prop_dictionary_remove(*prop_dictionary_t dict*, *const char *key*);

prop_object_t
prop_dictionary_get_keysym(*prop_dictionary_t dict*,
prop_dictionary_keysym_t keysym);

bool
prop_dictionary_set_keysym(*prop_dictionary_t dict*,
prop_dictionary_keysym_t keysym, *prop_object_t obj*);

void
prop_dictionary_remove_keysym(*prop_dictionary_t dict*,
prop_dictionary_keysym_t keysym);

bool
prop_dictionary_equals(*prop_dictionary_t dict1*, *prop_dictionary_t dict2*);

*const char **
prop_dictionary_keysym_cstring_nocopy(*prop_dictionary_keysym_t sym*);

bool
prop_dictionary_keysym_equals(*prop_dictionary_keysym_t keysym1*,
prop_dictionary_keysym_t keysym2);

*char **
prop_dictionary_externalize(*prop_dictionary_t dict*);

prop_dictionary_t
prop_dictionary_internalize(*const char *xml*);

bool
prop_dictionary_externalize_to_file(*prop_dictionary_t dict*,
*const char *path*);

prop_dictionary_t
prop_dictionary_internalize_from_file(*const char *path*);

DESCRIPTION

The **prop_dictionary** family of functions operate on the dictionary property collection object type. A dictionary is an unordered set of objects stored as key-value pairs.

prop_dictionary_create(*void*)
: Create an empty dictionary. The dictionary initially has no capacity. Returns NULL on failure.

prop_dictionary_create_with_capacity(*unsigned int capacity*)
: Create a dictionary with the capacity to store *capacity* objects. Returns NULL on failure.

prop_dictionary_copy(*prop_dictionary_t dict*)
: Copy a dictionary. The new dictionary has an initial capacity equal to the number of objects stored in the dictionary being copied. The new dictionary contains references to the original dictionary's objects, not copies of those objects (i.e. a shallow copy is made). If the original dictionary is immutable, the resulting dictionary is also immutable.

prop_dictionary_copy_mutable(*prop_dictionary_t dict*)
: Like **prop_dictionary_copy**(), except the resulting dictionary is always mutable.

prop_dictionary_count(*prop_dictionary_t dict*)
Returns the number of objects stored in the dictionary.

prop_dictionary_ensure_capacity(*prop_dictionary_t dict*)
Ensure that the dictionary has a total capacity of *capacity*, including objects already stored in the dictionary. Returns `true` if the capacity of the dictionary is greater or equal to *capacity* or if the expansion of the dictionary's capacity was successful and `false` otherwise. If the supplied object isn't a dictionary, `false` is returned.

prop_dictionary_iterator(*prop_dictionary_t dict*)
Create an iterator for the dictionary. The dictionary is retained by the iterator. A dictionary iterator returns the key symbols used to look up objects stored in the dictionary; to get the object itself, a dictionary lookup using this key symbol must be performed. Storing to or removing from the dictionary invalidates any active iterators for the dictionary. Returns `NULL` on failure.

prop_dictionary_all_keys(*prop_dictionary_t dict*)
Return an array of all of the dictionary key symbols (prop_dictionary_keysym_t) in the dictionary. This provides a way to iterate over the items in the dictionary while retaining the ability to mutate the dictionary; instead of iterating over the dictionary itself, iterate over the array of keys. The caller is responsible for releasing the array. Returns `NULL` on failure.

prop_dictionary_make_immutable(*prop_dictionary_t dict*)
Make *dict* immutable.

prop_dictionary_mutable(*prop_dictionary_t dict*)
Returns `true` if the dictionary is mutable.

prop_dictionary_get(*prop_dictionary_t dict*, *const char *key*)
Return the object stored in the dictionary with the key *key*. If no object is stored with the specified key, `NULL` is returned.

prop_dictionary_set(*prop_dictionary_t dict*, *const char *key*, *prop_object_t obj*)
Store a reference to the object *obj* with the key *key*. The object will be retained by the dictionary. If the key already exists in the dictionary, the object associated with that key will be released and replaced with the new object. Returns `true` if storing the object was successful and `false` otherwise.

prop_dictionary_remove(*prop_dictionary_t dict*, *const char *key*)
Remove the reference to the object stored in the dictionary with the key *key*. The object will be released.

prop_dictionary_get_keysym(*prop_dictionary_t dict*, *prop_dictionary_keysym_t sym*)
Like **prop_dictionary_get**(), but the lookup is performed using a key symbol returned by a dictionary iterator. The results are undefined if the iterator used to obtain the key symbol is not associated with *dict*.

prop_dictionary_set_keysym(*prop_dictionary_t dict*, *prop_dictionary_keysym_t sym*, *prop_object_t obj*)
Like **prop_dictionary_set**(), but the lookup of the object to replace is performed using a key symbol returned by a dictionary iterator. The results are undefined if the iterator used to obtain the key symbol is not associated with *dict*.

prop_dictionary_remove_keysym(*prop_dictionary_t dict*, *prop_dictionary_keysym_t sym*)
Like **prop_dictionary_remove**(), but the lookup of the object to remove is performed using a

key symbol returned by a dictionary iterator. The results are undefined if the iterator used to obtain the key symbol is not associated with *dict*.

prop_dictionary_equals(*prop_dictionary_t dict1*, *prop_dictionary_t dict2*)
Returns `true` if the two dictionaries are equivalent. Note: Objects contained in the dictionary are compared by value, not by reference.

prop_dictionary_keysym_cstring_nocopy(*prop_dictionary_keysym_t keysym*)
Returns an immutable reference to the dictionary key symbol's string value.

prop_dictionary_keysym_equals(*prop_dictionary_keysym_t keysym1*, *prop_dictionary_keysym_t keysym2*)
Returns `true` if the two dictionary key symbols are equivalent.

prop_dictionary_externalize(*prop_dictionary_t dict*)
Externalizes a dictionary, returning a NUL-terminated buffer containing the XML representation of the dictionary. The caller is responsible for freeing the returned buffer. If converting to the external representation fails for any reason, `NULL` is returned.

In user space, the buffer is allocated using `malloc`(3). In the kernel, the buffer is allocated using `malloc`(9) using the malloc type `M_TEMP`.

prop_dictionary_internalize(*const char *xml*)
Parse the XML representation of a property list in the NUL-terminated buffer *xml* and return the corresponding dictionary. Returns `NULL` if parsing fails for any reason.

prop_dictionary_externalize_to_file(*prop_dictionary_t dict*, *const char *path*)
Externalizes a dictionary and writes it to the file specified by *path*. The file is saved with the mode `0666` as modified by the process's file creation mask (see `umask`(3)) and is written atomically. Returns `false` if externalizing or writing the dictionary fails for any reason.

prop_dictionary_internalize_from_file(*const char *path*)
Reads the XML property list contained in the file specified by *path*, internalizes it, and returns the corresponding array. Returns `NULL` on failure.

SEE ALSO

`prop_array`(3), `prop_bool`(3), `prop_data`(3), `prop_dictionary_util`(3), `prop_number`(3), `prop_object`(3), `prop_string`(3), `proplib`(3)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

NAME

prop_dictionary_util, **prop_dictionary_get_bool**, **prop_dictionary_set_bool**, **prop_dictionary_get_int8**, **prop_dictionary_get_uint8**, **prop_dictionary_set_int8**, **prop_dictionary_set_uint8**, **prop_dictionary_get_int16**, **prop_dictionary_get_uint16**, **prop_dictionary_set_int16**, **prop_dictionary_set_uint16**, **prop_dictionary_get_int32**, **prop_dictionary_get_uint32**, **prop_dictionary_set_int32**, **prop_dictionary_set_uint32**, **prop_dictionary_get_int64**, **prop_dictionary_get_uint64**, **prop_dictionary_set_int64**, **prop_dictionary_set_uint64**, **prop_dictionary_get_cstring**, **prop_dictionary_set_cstring**, **prop_dictionary_get_cstring_nocopy**, **prop_dictionary_set_cstring_nocopy**

LIBRARY

library “libprop”

SYNOPSIS

```
#include <prop/proplib.h>

bool
prop_dictionary_get_bool(prop_dictionary_t dict, const char *key,
    bool *valp);

bool
prop_dictionary_set_bool(prop_dictionary_t dict, const char *key,
    bool val);

bool
prop_dictionary_get_int8(prop_dictionary_t dict, const char *key,
    int8_t *valp);

bool
prop_dictionary_get_uint8(prop_dictionary_t dict, const char *key,
    uint8_t *valp);

bool
prop_dictionary_set_int8(prop_dictionary_t dict, const char *key,
    int8_t val);

bool
prop_dictionary_set_uint8(prop_dictionary_t dict, const char *key,
    uint8_t val);

bool
prop_dictionary_get_int16(prop_dictionary_t dict, const char *key,
    int16_t *valp);

bool
prop_dictionary_get_uint16(prop_dictionary_t dict, const char *key,
    uint16_t *valp);

bool
prop_dictionary_set_int16(prop_dictionary_t dict, const char *key,
    int16_t val);
```

bool
prop_dictionary_set_uint16(*prop_dictionary_t dict*, *const char *key*, *uint16_t val*);

bool
prop_dictionary_get_int32(*prop_dictionary_t dict*, *const char *key*, *int32_t *valp*);

bool
prop_dictionary_get_uint32(*prop_dictionary_t dict*, *const char *key*, *uint32_t *valp*);

bool
prop_dictionary_set_int32(*prop_dictionary_t dict*, *const char *key*, *int32_t val*);

bool
prop_dictionary_set_uint32(*prop_dictionary_t dict*, *const char *key*, *uint32_t val*);

bool
prop_dictionary_get_int64(*prop_dictionary_t dict*, *const char *key*, *int64_t *valp*);

bool
prop_dictionary_get_uint64(*prop_dictionary_t dict*, *const char *key*, *uint64_t *valp*);

bool
prop_dictionary_set_int64(*prop_dictionary_t dict*, *const char *key*, *int64_t val*);

bool
prop_dictionary_set_uint64(*prop_dictionary_t dict*, *const char *key*, *uint64_t val*);

bool
prop_dictionary_get_cstring(*prop_dictionary_t dict*, *const char *key*, *char **strp*);

bool
prop_dictionary_set_cstring(*prop_dictionary_t dict*, *const char *key*, *const char *str*);

bool
prop_dictionary_get_cstring_nocopy(*prop_dictionary_t dict*, *const char *key*, *const char **strp*);

bool
prop_dictionary_set_cstring_nocopy(*prop_dictionary_t dict*, *const char *key*, *const char *strp*);

DESCRIPTION

The **prop_dictionary_util** family of functions are provided to make getting and setting values in dictionaries more convenient in some applications.

The getters check the type of the returned object and, in some cases, also ensure that the returned value is within the range implied by the getter's value type.

The setters handle object creation and release for the caller.

The **prop_dictionary_get_cstring**() function returns dynamically allocated memory. See prop_string(3) for more information.

The **prop_dictionary_get_cstring_nocopy**() and **prop_dictionary_set_cstring_nocopy**() functions do not copy the string that is set or returned. See prop_string(3) for more information.

RETURN VALUES

The **prop_dictionary_util** getter functions return true if the object exists in the dictionary and the value is in-range, or false otherwise.

The **prop_dictionary_util** setter functions return true if creating the object and storing it in the dictionary is successful, or false otherwise.

SEE ALSO

prop_bool(3), prop_dictionary(3), prop_number(3), proplib(3)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

NAME

prop_ingest_context_alloc, **prop_ingest_context_free**, **prop_ingest_context_error**, **prop_ingest_context_type**, **prop_ingest_context_key**, **prop_ingest_context_private**, **prop_dictionary_ingest** — Ingest a dictionary into an arbitrary binary format

SYNOPSIS

```
#include <prop/proplib.h>

prop_ingest_context_t
prop_ingest_context_alloc(void *private);

void
prop_ingest_context_free(prop_ingest_context_t ctx);

prop_ingest_error_t
prop_ingest_context_error(prop_ingest_context_t ctx);

prop_type_t
prop_ingest_context_type(prop_ingest_context_t ctx);

const char *
prop_ingest_context_key(prop_ingest_context_t ctx);

void *
prop_ingest_context_private(prop_ingest_context_t ctx);

bool
prop_dictionary_ingest(prop_dictionary_t dict,
    const prop_ingest_table_entry rules[], prop_ingest_context_t ctx);

typedef bool
(*prop_ingest_handler_t)(prop_ingest_context_t, prop_object_t);
```

DESCRIPTION

The **prop_dictionary_ingest** function provides a convenient way to convert a property list into an arbitrary binary format or to extract values from dictionaries in a way that is convenient to an application (for configuration files, for example).

prop_dictionary_ingest is driven by a table of rules provided by the application. Each rule consists of three items:

- A C string containing a key to be looked up in the dictionary.
- The expected property type of the object associated with the key (or `PROP_TYPE_UNKNOWN` to specify that any type is allowed).
- A callback function of type `prop_ingest_handler_t` that will perform the translation for the application.

The table is constructed using a series of macros as follows:

```
static const prop_ingest_table_entry ingest_rules[] = {
        PROP_INGEST("file-name", PROP_TYPE_STRING, ingest_filename),
        PROP_INGEST("count", PROP_TYPE_NUMBER, ingest_count),
        PROP_INGEST_OPTIONAL("required", PROP_TYPE_BOOL, ingest_required),
        PROP_INGEST_OPTIONAL("extra", PROP_TYPE_UNKNOWN, ingest_extra),
        PROP_INGEST_END
};
```

The PROP_INGEST macro specifies that the key is required to be present in the dictionary. The PROP_INGEST_OPTIONAL macro specifies that the presence of the key in the dictionary is optional. The PROP_INGEST_END macro marks the end of the rules table.

In each case, **prop_dictionary_ingest** looks up the rule's key in the dictionary. If an object is present in the dictionary at that key, its type is checked against the type specified in the rule. A type specification of PROP_TYPE_UNKNOWN allows the object to be of any type. If the object does not exist and the rule is not marked as optional, then an error is returned. Otherwise, the handler specified in the rule is invoked with the ingest context and the object (or NULL if the key does not exist in the dictionary). The handler should return false if the value of the object is invalid to indicate failure and true otherwise.

The ingest context contains several pieces of information that are useful during the ingest process. The context also provides specific error information should the ingest fail.

prop_ingest_context_alloc(*void *private*)
: Allocate an ingest context. The argument *private* may be used to pass application-specific context to the ingest handlers. Note that an ingest context can be re-used to perform multiple ingests. Returns NULL on failure.

prop_ingest_context_free(*prop_ingest_context_t ctx*)
: Free an ingest context.

prop_ingest_context_error(*prop_ingest_context_t ctx*)
: Returns the code indicating the error encountered during ingest. The following error codes are defined:

PROP_INGEST_ERROR_NO_ERROR	No error was encountered during ingest.
PROP_INGEST_ERROR_NO_KEY	A non-optional key was not found in the dictionary.
PROP_INGEST_ERROR_WRONG_TYPE	An object in the dictionary was not the same type specifed in the rules.
PROP_INGEST_ERROR_HANDLER_FAILED	An object's handler returned false.

prop_ingest_context_type(*prop_ingest_context_t ctx*)
: Returns the type of the last object visited during an ingest. When called by an ingest handler, it returns the type of the object currently being processed.

prop_ingest_context_key(*prop_ingest_context_t ctx*)
: Returns the last dictionary key looked up during an ingest. When called by an ingest handler, it returns the dictionary key corresponding to the object currently being processed.

prop_ingest_context_private(*prop_ingest_context_t ctx*)
: Returns the private data set when the context was allocated with **prop_ingest_context_alloc**().

SEE ALSO

prop_dictionary(3), proplib(3)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

NAME

prop_number, **prop_number_create_integer**, **prop_number_create_unsigned_integer**, **prop_number_copy**, **prop_number_size**, **prop_number_unsigned**, **prop_number_integer_value**, **prop_number_unsigned_integer_value**, **prop_number_equals**, **prop_number_equals_integer**, **prop_number_equals_unsigned_integer** — numeric value property object

LIBRARY

library “libprop”

SYNOPSIS

```
#include <prop/proplib.h>

prop_number_t
prop_number_create_integer(int64_t val);

prop_number_t
prop_number_create_unsigned_integer(uint64_t val);

prop_number_t
prop_number_copy(prop_number_t number);

int
prop_number_size(prop_number_t number);

bool
prop_number_unsigned(prop_number_t number);

int64_t
prop_number_integer_value(prop_number_t number);

uint64_t
prop_number_unsigned_integer_value(prop_number_t number);

bool
prop_number_equals(prop_number_t num1, prop_number_t num2);

bool
prop_number_equals_integer(prop_number_t number, int64_t val);

bool
prop_number_equals_unsigned_integer(prop_number_t number, uint64_t val);
```

DESCRIPTION

The **prop_number** family of functions operate on a numeric value property object type. Values are either signed or unsigned, and promoted to a 64-bit type (int64_t or uint64_t, respectively).

It is possible to compare number objects that differ in sign. Such comparisons first test to see if each object is within the valid number range of the other:

- Signed numbers that are greater than or equal to 0 can be compared to unsigned numbers.
- Unsigned numbers that are less than or equal to the largest signed 64-bit value (INT64_MAX) can be compared to signed numbers.

Number objects have a different externalized representation depending on their sign:

- Signed numbers are externalized in base-10 (decimal).
- Unsigned numbers are externalized in base-16 (hexadecimal).

When numbers are internalized, the sign of the resulting number object (and thus its valid range) is determined by a set of rules evaluated in the following order:

- If the first character of the number is a '-' then the number is signed.
- If the first two characters of the number are '0x' then the number is unsigned.
- If the number value fits into the range of a signed number then the number is signed.
- In all other cases, the number is unsigned.

prop_number_create_integer(*int64_t val*)
Create a numeric value object with the signed value *val*. Returns NULL on failure.

prop_number_create_unsigned_integer(*uint64_t val*)
Create a numeric value object with the unsigned value *val*. Returns NULL on failure.

prop_number_copy(*prop_number_t number*)
Copy a numeric value object. If the supplied object isn't a numeric value, NULL is returned.

prop_number_size(*prop_number_t number*)
Returns 8, 16, 32, or 64, representing the number of bits required to hold the value of the object. If the supplied object isn't a numeric value, NULL is returned.

prop_number_unsigned(*prop_number_t number*)
Returns true if the numeric value object has an unsigned value.

prop_number_integer_value(*prop_number_t number*)
Returns the signed integer value of the numeric value object. If the supplied object isn't a numeric value, zero is returned. Thus, it is not possible to distinguish between ''not a prop_number_t'' and ''prop_number_t has a value of 0''.

prop_number_unsigned_integer_value(*prop_number_t number*)
Returns the unsigned integer value of the numeric value object. If the supplied object isn't a numeric value, zero is returned. Thus, it is not possible to distinguish between ''not a prop_number_t'' and ''prop_number_t has a value of 0''.

prop_number_equals(*prop_number_t num1*, *prop_number_t num2*)
Returns true if the two numeric value objects are equivalent. If at least one of the supplied objects isn't a numeric value, false is returned.

prop_number_equals_integer(*prop_number_t number*, *int64_t val*)
Returns true if the object's value is equivalent to the signed value *val*. If the supplied object isn't a numerical value or if *val* exceeds INT64_MAX, false is returned.

prop_number_equals_unsigned_integer(*prop_number_t number*, *uint64_t val*)
Returns true if the object's value is equivalent to the unsigned value *val*. If the supplied object isn't a numerical value or if *val* exceeds INT64_MAX, false is returned.

SEE ALSO

prop_array(3), prop_bool(3), prop_data(3), prop_dictionary(3), prop_object(3), prop_string(3), proplib(3)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

NAME

prop_object, **prop_object_retain**, **prop_object_release**, **prop_object_type**, **prop_object_equals**, **prop_object_iterator_next**, **prop_object_iterator_reset**, **prop_object_iterator_release** — general property container object functions

LIBRARY

library “libprop”

SYNOPSIS

```
#include <prop/proplib.h>

void
prop_object_retain(prop_object_t obj);

void
prop_object_release(prop_object_t obj);

prop_type_t
prop_object_type(prop_object_t obj);

bool
prop_object_equals(prop_object_t obj1, prop_object_t obj2);

prop_object_t
prop_object_iterator_next(prop_object_iterator_t iter);

void
prop_object_iterator_reset(prop_object_iterator_t iter);

void
prop_object_iterator_release(prop_object_iterator_t iter);
```

DESCRIPTION

The **prop_object** family of functions operate on all property container object types.

prop_object_retain(*prop_object_t obj*)
Increment the reference count on an object.

prop_object_release(*prop_object_t obj*)
Decrement the reference count on an object. If the last reference is dropped, the object is freed.

prop_object_type(*prop_object_t obj*)
Determine the type of the object. Objects are one of the following types:

PROP_TYPE_BOOL	Boolean value (prop_bool_t)
PROP_TYPE_NUMBER	Number (prop_number_t)
PROP_TYPE_STRING	String (prop_string_t)
PROP_TYPE_DATA	Opaque data (prop_data_t)
PROP_TYPE_ARRAY	Array (prop_array_t)
PROP_TYPE_DICTIONARY	Dictionary (prop_dictionary_t)
PROP_TYPE_DICT_KEYSYM	Dictionary key symbol (prop_dictionary_keysym_t)

If *obj* is NULL, then PROP_TYPE_UNKNOWN is returned.

prop_object_equals(*prop_object_t obj1*, *prop_object_t obj2*)
Returns true if the two objects are of the same type and are equivalent.

prop_object_iterator_next(*prop_object_iterator_t iter*)
Return the next object in the collection (array or dictionary) being iterated by the iterator *iter*. If there are no more objects in the collection, NULL is returned.

prop_object_iterator_reset(*prop_object_iterator_t iter*)
Reset the iterator to the first object in the collection being iterated by the iterator *iter*.

prop_object_iterator_release(*prop_object_iterator_t iter*)
Release the iterator *iter*.

SEE ALSO

prop_array(3), prop_bool(3), prop_data(3), prop_dictionary(3), prop_number(3), prop_string(3), proplib(3)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

NAME

prop_array_send_ioctl, **prop_array_recv_ioctl**, **prop_dictionary_send_ioctl**, **prop_dictionary_recv_ioctl**, **prop_dictionary_sendrecv_ioctl** — Send and receive propertly lists to and from the kernel using ioctl

SYNOPSIS

```
#include <prop/proplib.h>

int
prop_array_send_ioctl(prop_array_t array, int fd, unsigned long cmd);

int
prop_array_recv_ioctl(int fd, unsigned long cmd, prop_array_t *arrayp);

int
prop_dictionary_send_ioctl(prop_dictionary_t dict, int fd,
    unsigned long cmd);

int
prop_dictionary_recv_ioctl(int fd, unsigned long cmd,
    prop_dictionary_t *dictp);

prop_dictionary_sendrecv_ioctl(prop_dictionary_t dict, int fd,
    unsigned long cmd, prop_dictionary_t *dictp);
```

DESCRIPTION

The **prop_array_send_ioctl**, **prop_array_recv_ioctl**, **prop_dictionary_send_ioctl**, **prop_dictionary_recv_ioctl**, and **prop_dictionary_sendrecv_ioctl** functions implement the user space side of a protocol for sending property lists to and from the kernel using ioctl(2).

RETURN VALUES

If successful, functions return zero. Otherwise, an error number is returned to indicate the error.

ERRORS

prop_array_send_ioctl() and **prop_dictionary_send_ioctl**() will fail if:

[ENOMEM] Cannot allocate memory

[ENOTSUP] Not supported

prop_array_recv_ioctl() and **prop_dictionary_recv_ioctl**() will fail if:

[EIO] Input/output error

[ENOTSUP] Not supported

In addition to these, ioctl(2) errors may be returned.

EXAMPLES

The following (simplified) example demonstrates using **prop_dictionary_send_ioctl**() and **prop_dictionary_recv_ioctl**() in an application:

```
void
foo_setprops(prop_dictionary_t dict)
{
    int fd;
```

```
    fd = open("/dev/foo", O_RDWR, 0640);
    if (fd == -1)
        return;

    (void) prop_dictionary_send_ioctl(dict, fd, FOOSETPROPS);

    (void) close(fd);
}

prop_dictionary_t
foo_getprops(void)
{
    prop_dictionary_t dict;
    int fd;

    fd = open("/dev/foo", O_RDONLY, 0640);
    if (fd == -1)
        return (NULL);

    if (prop_dictionary_recv_ioctl(fd, FOOGETPROPS, &dict) != 0)
        return (NULL);

    (void) close(fd);

    return (dict);
}
```

The **prop_dictionary_sendrecv_ioctl** function combines the send and receive functionality, allowing for ioctls that require two-way communication (for example to specify arguments for the ioctl operation).

SEE ALSO

prop_array(3), prop_dictionary(3), proplib(3), prop_copyin_ioctl(9)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

NAME

prop_string, **prop_string_create**, **prop_string_create_cstring**, **prop_string_create_cstring_nocopy**, **prop_string_copy**, **prop_string_copy_mutable**, **prop_string_size**, **prop_string_mutable**, **prop_string_cstring**, **prop_string_cstring_nocopy**, **prop_string_append**, **prop_string_append_cstring**, **prop_string_equals**, **prop_string_equals_cstring** — string value property object

LIBRARY

library “libprop”

SYNOPSIS

```
#include <prop/proplib.h>

prop_string_t
prop_string_create(void);

prop_string_t
prop_string_create_cstring(const char *cstring);

prop_string_t
prop_string_create_cstring_nocopy(const char *cstring);

prop_string_t
prop_string_copy(prop_string_t string);

prop_string_t
prop_string_copy_mutable(prop_string_t string);

char *
prop_string_cstring(prop_string_t string);

const char *
prop_string_cstring_nocopy(prop_string_t string);

bool
prop_string_append(prop_string_t str1, prop_string_t str2);

bool
prop_string_append_cstring(prop_string_t string, const char *cstring);

bool
prop_string_equals(prop_string_t str1, prop_string_t str2);

bool
prop_string_equals_cstring(prop_string_t string, const char *cstring);
```

DESCRIPTION

The **prop_string** family of functions operate on a string value property object type.

prop_string_create(*void*)
: Create an empty mutable string. Returns NULL on failure.

prop_string_create_cstring(*const char *cstring*)
: Create a mutable string that contains a copy of *cstring*. Returns NULL on failure.

prop_string_create_cstring_nocopy(*const char *cstring*)
: Create an immutable string that contains a reference to *cstring*. Returns NULL on failure.

prop_string_copy(*prop_string_t string*)
Copy a string. If the the string being copied is an immutable external C string reference, then the copy is also immutable and references the same external C string. Returns NULL on failure.

prop_string_copy_mutable(*prop_string_t string*)
Copy a string, always creating a mutable copy. Returns NULL on failure.

prop_string_size(*prop_string_t string*)
Returns the size of the string, not including the terminating NUL. If the supplied object isn't a string, zero is returned.

prop_string_mutable(*prop_string_t string*)
Returns true if the string is mutable. If the supplied object isn't a string, false is returned.

prop_string_cstring(*prop_string_t string*)
Returns a copy of the string's contents as a C string. The caller is responsible for freeing the returned buffer.

In user space, the buffer is allocated using malloc(3). In the kernel, the buffer is allocated using malloc(9) using the malloc type M_TEMP.

Returns NULL on failure.

prop_string_cstring_nocopy(*prop_string_t string*)
Returns an immutable reference to the contents of the string as a C string. If the supplied object isn't a string, NULL is returned.

prop_string_append(*prop_string_t str1*, *prop_string_t str2*)
Append the contents of *str2* to *str1*, which must be mutable. Returns true upon success and false otherwise.

prop_string_append_cstring(*prop_string_t string*, *const char *cstring*)
Append the C string *cstring* to *string*, which must be mutable. Returns true upon success and false otherwise.

prop_string_equals(*prop_string_t str1*, *prop_string_t str2*)
Returns true if the two string objects are equivalent.

prop_string_equals_cstring(*prop_string_t string*, *const char *cstring*)
Returns true if the string's value is equivalent to *cstring*.

SEE ALSO

prop_array(3), prop_bool(3), prop_data(3), prop_dictionary(3), prop_number(3), prop_object(3), proplib(3)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

NAME

proplib — property container object library

LIBRARY

library “libprop”

SYNOPSIS

```
#include <prop/proplib.h>
```

DESCRIPTION

The **proplib** library provides an abstract interface for creating and manipulating property lists. Property lists have object types for boolean values, opaque data, numbers, and strings. Structure is provided by the array and dictionary collection types.

Property lists can be passed across protection boundaries by translating them to an external representation. This external representation is an XML document whose format is described by the following DTD:

```
http://www.apple.com/DTDs/PropertyList-1.0.dtd
```

Property container objects are reference counted. When an object is created, its reference count is set to 1. Any code that keeps a reference to an object, including the collection types (arrays and dictionaries), must “retain” the object (increment its reference count). When that reference is dropped, the object must be “released” (reference count decremented). When an object’s reference count drops to 0, it is automatically freed.

The rules for managing reference counts are very simple:

- If you create an object and do not explicitly maintain a reference to it, you must release it.
- If you get a reference to an object from other code and wish to maintain a reference to it, you must retain the object. You are responsible for releasing the object once you drop that reference.
- You must never release an object unless you create it or retain it.

Object collections may be iterated by creating a special iterator object. Iterator objects are special; they may not be retained, and they are released using an iterator-specific release function.

SEE ALSO

prop_array(3), prop_bool(3), prop_data(3), prop_dictionary(3), prop_dictionary_util(3), prop_number(3), prop_object(3), prop_send_ioctl(3), prop_string(3)

HISTORY

The **proplib** property container object library first appeared in NetBSD 4.0.

CAVEATS

proplib does not have a ‘date’ object type, and thus will not parse ‘date’ elements from an Apple XML property list.

The **proplib** ‘number’ object type differs from the Apple XML property list format in the following ways:

- The external representation is in base 16, not base 10. **proplib** is able to parse base 8, base 10, and base 16 ‘integer’ elements.
- Internally, integers are always stored as unsigned numbers (uint64_t). Therefore, the external representation will never be negative.

- Because floating point numbers are not supported, ‘real’ elements from an Apple XML property list will not be parsed.

In order to facilitate use of **proplib** in kernel, standalone, and user space environments, the **proplib** parser is not a real XML parser. It is hard-coded to parse only the property list external representation.

NAME

pset_create, **pset_assign**, **pset_bind**, **pset_destroy** — processor sets

SYNOPSIS

```
#include <sys/pset.h>

int
pset_create(psetid_t *psid);

int
pset_assign(psetid_t psid, cpuid_t cpuid, psetid_t *opsid);

int
pset_bind(psetid_t psid, idtype_t type, id_t id, psetid_t *opsid);

int
pset_destroy(psetid_t psid);
```

DESCRIPTION

The processor sets API provides the possibility to bind processes or threads to specific processors or groups of processors. This section describes the functions used to control processor sets.

FUNCTIONS

pset_create(*psid*)

Creates a processor set, and returns its ID into *psid*.

pset_assign(*psid*, *cpu*, *opsid*)

Assigns the processor specified by *cpuid* to the processor set specified by *psid*. Stores the current processor set ID of the processor or PS_NONE into *opsid*, if the pointer is not NULL.

If *psid* is set to PS_QUERY, then the current processor set ID will be returned into *psid*, and no assignment will be performed.

If *psid* is set to PS_MYID, then the processor set ID of the calling process will be used, and *psid* will be ignored.

If *psid* is set to PS_NONE, any assignment to the processor will be cleared.

pset_bind(*psid*, *type*, *id*, *opsid*)

Binds the target specified by *id* to the processor set specified by *psid*. The current processor set ID to which the target is bound or PS_NONE will be returned in *opsid*, if the pointer is not NULL. NetBSD supports the following types of targets specified by *type*:

P_PID Process identified by the PID.

P_LWPID
Thread of the calling process indentified by the LID.

If *psid* is set to PS_QUERY, then the current processor set ID to which the target is bound or PS_NONE will be returned in *opsid*, and no binding will be performed. If *psid* is set to PS_MYID, then the processor set ID of the calling process will be used.

If *psid* is set to PS_NONE, the specified target will be unbound from the processor set.

pset_destroy(*psid*)

Destroys the processor set specified by *psid*. Before destroying the processor set, all related assignments of the processors will be cleared, and all bound threads will be unbound.

If *psid* is PS_MYID, the processor set ID of the caller thread will be used.

NOTES

The *pset_bind* function can return the current processor set ID to which the target is bound, or PS_NONE. However, for example, the process may have many threads, which could be bound to different processor sets. In such a case it is unspecified which thread will be used to return the information.

RETURN VALUES

Upon successful completion these functions return 0. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

The **pset_create**() function fails if:

[ENOMEM] No memory is available for creation of the processor set, or limit of the allowed count of the processor sets was reached.

[EPERM] The calling process is not the super-user.

The **pset_assign**() function fails if:

[EBUSY] Another operation is performing on the processor set.

[EINVAL] *psid* or *cpuid* are invalid.

[EPERM] The calling process is not the super-user, and *psid* is not PS_QUERY.

The **pset_bind**() function fails if:

[EBUSY] Another operation is performing on the processor set.

[EINVAL] *psid* or *type* are invalid.

[EPERM] The calling process is not the super-user, and *psid* is not PS_QUERY.

[ESRCH] The specified target was not found.

The **pset_destroy**() function fails if:

[EBUSY] Another operation is performing on the processor set.

[EPERM] The calling process is not the super-user.

SEE ALSO

sched(3), schedctl(8)

STANDARDS

This API is expected to be compatible with the APIs found in Solaris and HP-UX operating systems.

HISTORY

The processor sets appeared in NetBSD 5.0.

NAME

psignal, **sys_siglist**, **sys_signame** — system signal messages

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

void
psignal(unsigned sig, const char *s);

extern const char * const sys_siglist[];
extern const char * const sys_signame[];
```

DESCRIPTION

The **psignal**() function locates the descriptive message string for the given signal number *sig* and writes it to the standard error.

If the argument *s* is non-NULL it is written to the standard error file descriptor prior to the message string, immediately followed by a colon and a space. If the signal number is not recognized (sigaction(2)), the string “Unknown signal” is produced.

The message strings can be accessed directly using the external array *sys_siglist*, indexed by recognized signal numbers. The external array *sys_signame* is used similarly and contains short, upper-case abbreviations for signals which are useful for recognizing signal names in user input. The defined variable NSIG contains a count of the strings in *sys_siglist* and *sys_signame*.

SEE ALSO

sigaction(2), perror(3), setlocale(3), strsignal(3)

HISTORY

The **psignal**() function appeared in 4.2BSD.

<<<<<<< pthread.3 ======= >>>>>>> 1.8

NAME

pthread — POSIX Threads Library

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

cc [flags] files -lpthread [libraries]
```

DESCRIPTION

The **pthread** library provides an implementation of the standard POSIX threads framework.

Note that the system private thread interfaces upon which the **pthread** library is built are subject to change without notice. In order to remain compatible with future NetBSD releases, programs must be linked against the dynamic version of the thread library. Statically linked programs using the POSIX threads framework may not work when run on a future version of the system.

ENVIRONMENT

PTHREAD_CONCURRENCY
: The current version of the system does not inspect this variable. It is reserved for use by the **pthread** library.

PTHREAD_DIAGASSERT
: Possible values are any combinations of:

  A Ignore errors.

  a Abort on errors, creating a core dump for further debugging.

  E Do not log errors to stdout.

  e Log errors to stdout.

  L Do not log errors via syslogd(8).

  l Log errors via syslogd(8).

PTHREAD_RRTIME
: The current version of the system does not inspect this variable. It is reserved for use by the **pthread** library.

PTHREAD_STACKSIZE
: Integer value giving the stack size in kilobytes. This allows to set a smaller stack size than the default stack size. The default stack size is the current limit on the stack size as set with the shell's command to change limits (**limit** for csh(1), or **ulimit** for sh(1)).

SEE ALSO

pthread_attr(3), pthread_barrier_destroy(3), pthread_barrier_init(3), pthread_barrier_wait(3), pthread_barrierattr(3), pthread_cancel(3), pthread_cleanup_push(3), pthread_cond_broadcast(3), pthread_cond_destroy(3), pthread_cond_init(3), pthread_cond_wait(3), pthread_condattr(3), pthread_create(3), pthread_detach(3), pthread_equal(3), pthread_exit(3), pthread_getspecific(3), pthread_join(3), pthread_key_create(3), pthread_key_delete(3), pthread_kill(3), pthread_mutex_destroy(3), pthread_mutex_init(3), pthread_mutex_lock(3), pthread_mutex_unlock(3), pthread_mutexattr(3), pthread_once(3), pthread_rwlock_destroy(3), pthread_rwlock_init(3), pthread_rwlock_rdlock(3), pthread_rwlock_unlock(3), pthread_rwlock_wrlock(3), pthread_rwlockattr(3), pthread_schedparam(3), pthread_self(3), pthread_setspecific(3), pthread_sigmask(3), pthread_spin_destroy(3), pthread_spin_init(3), pthread_spin_lock(3), pthread_spin_unlock(3), pthread_testcancel(3)

STANDARDS

The **pthread** library conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

pthread_atfork — register handlers to be called when process forks

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <pthread.h>

int
pthread_atfork(void (*prepare)(void), void (*parent)(void),
    void (*child)(void));
```

DESCRIPTION

The **pthread_atfork**() function registers the provided handler functions to be called when the fork(2) function is called. Each of the three handlers is called at a different place in the fork(2) sequence. The *prepare* handler is called in the parent process before the fork happens, the *parent* handler is called in the parent process after the fork has happened, and the *child* handler is called in the child process after the fork has happened. The *parent* and *child* handlers are called in the order in which they were registered, while the *prepare* handlers are called in reverse of the order in which they were registered.

Any of the handlers given may be NULL.

The intended use of **pthread_atfork**() is to provide a consistent state to a child process from a multi-threaded parent process where locks may be acquired and released asynchronously with respect to the fork(2) call. Each subsystem with locks that are used in a child process should register handlers with **pthread_atfork**() that acquires those locks in the *prepare* handler and releases them in the *parent* handler.

RETURN VALUES

The **pthread_atfork**() function returns 0 on success and an error number on failure.

ERRORS

The following error code may be returned:

[ENOMEM] Insufficient memory exists to register the fork handlers.

SEE ALSO

fork(2)

STANDARDS

The **pthread_atfork**() function conforms to IEEE Std 1003.1c-1995 (“POSIX.1”).

HISTORY

The **pthread_atfork**() function first appeared in NetBSD 2.0.

CAVEATS

After calling fork(2) from a multithreaded process, it is only safe to call async-signal-safe functions until calling one of the exec(3) functions. The **pthread_***() functions are not async-signal-safe, so it is not safe to use such functions in the *child* handler.

BUGS

There is no way to unregister a handler registered with **pthread_atfork**().

NAME

pthread_attr_init, **pthread_attr_destroy**, **pthread_attr_setdetachstate**, **pthread_attr_getdetachstate**, **pthread_attr_setschedparam**, **pthread_attr_getschedparam** — thread attribute operations

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_attr_init(pthread_attr_t *attr);

int
pthread_attr_destroy(pthread_attr_t *attr);

int
pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

int
pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);

int
pthread_attr_setschedparam(pthread_attr_t * restrict attr,
    const struct sched_param * restrict param);

int
pthread_attr_getschedparam(const pthread_attr_t * restrict attr,
    struct sched_param * restrict param);
```

DESCRIPTION

Thread attributes are used to specify parameters to **pthread_create**(). One attribute object can be used in multiple calls to **pthread_create**(), with or without modifications between calls.

The **pthread_attr_init**() function initializes *attr* with all the default thread attributes.

The **pthread_attr_destroy**() function destroys *attr*.

The **pthread_attr_set***() functions set the attribute that corresponds to each function name.

The **pthread_attr_get***() functions copy the value of the attribute that corresponds to each function name to the location pointed to by the second function parameter.

The attribute parameters for the **pthread_attr_setdetachstate**() and **pthread_attr_getdetachstate**() are mutually exclusive and must be one of:

PTHREAD_CREATE_JOINABLE
: The threads must explicitely be waited for using the **pthread_join**() function once they exit for their status to be received and their resources to be freed. This is the default.

PTHREAD_CREATE_DETACHED
: The thread's resources will automatically be freed once the thread exits, and the thread will not be joined.

RETURN VALUES

If successful, these functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

pthread_attr_init() shall fail if:

[ENOMEM] Out of memory.

pthread_attr_destroy() may fail if:

[EINVAL] The value specified by *attr* is invalid.

pthread_attr_setdetachstate() shall fail if:

[EINVAL] The value specified by *detachstate* is invalid.

pthread_attr_setschedparam() may fail if:

[EINVAL] The value specified by *attr* is invalid.

[ENOTSUP] The value specified by *param* is invalid.

SEE ALSO

pthread_create(3), pthread_join(3)

STANDARDS

pthread_attr_init(), **pthread_attr_destroy**(), **pthread_attr_setdetachstate**(), **pthread_attr_getdetachstate**(), **pthread_attr_setschedparam**(), and **pthread_attr_getschedparam**() conform to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

pthread_attr_getname_np — set descriptive name of an attribute

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <pthread.h>

int
pthread_attr_getname_np(const pthread_attr_t attr, char *name, size_t len);
```

DESCRIPTION

pthread_attr_getname_np() gets the descriptive name of the attribute. It takes the following arguments.

attr The attribute whose descriptive name will be obtained.

name The buffer to be filled with the descriptive name of the attribute.

len The size of the buffer *name* in bytes.

RETURN VALUES

pthread_attr_getname_np() returns 0 on success. Otherwise, **pthread_attr_getname_np**() returns an error number described in errno(2).

SEE ALSO

errno(2), pthread_attr_setname_np(3)

NAME

pthread_attr_setcreatesuspend_np — set attribute to create a thread suspended

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_attr_setcreatesuspend_np(pthread_attr_t attr);
```

DESCRIPTION

The **pthread_attr_setcreatesuspend_np**() function sets the *attr* argument, so that if this *attr* is used in a pthread_create(3) call, then the thread created will not run, but it will remain blocked in the suspended queue, until pthread_resume_np(3) is called on it.

RETURN VALUES

The **pthread_attr_setcreatesuspend_np**() function always returns 0.

ERRORS

pthread_attr_setcreatesuspend_np() never fails.

SEE ALSO

pthread_create(3), pthread_resume_np(3), pthread_suspend_np(3)

NAME

pthread_attr_setname_np — set descriptive name of an attribute

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <pthread.h>

int
pthread_attr_setname_np(pthread_attr_t attr, const char *name, void *arg);
```

DESCRIPTION

pthread_attr_setname_np() sets the descriptive name of the attribute. It takes the following arguments.

attr The attribute whose descriptive name will be set.

name The printf(3) format string to be used to construct the descriptive name of the attribute. The resulted descriptive name should be shorter than PTHREAD_MAX_NAMELEN_NP.

arg The printf(3) argument used with *name*.

RETURN VALUES

pthread_attr_setname_np() returns 0 on success. Otherwise, **pthread_attr_setname_np**() returns an error number described in errno(2).

SEE ALSO

errno(2), pthread_attr_getname_np(3)

NAME

pthread_barrier_destroy — destroy a barrier

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_barrier_destroy(pthread_barrier_t *barrier);
```

DESCRIPTION

The **pthread_barrier_destroy**() function causes the resources allocated to *barrier* to be released. No threads should be blocked on *barrier*.

RETURN VALUES

If successful, **pthread_barrier_destroy**() will return zero. Otherwise an error value will be returned.

ERRORS

pthread_barrier_destroy() may fail if:

[EBUSY] The *barrier* still has active threads associated with it.

[EINVAL] The value specified by *barrier* is invalid.

SEE ALSO

pthread_barrier_init(3), pthread_barrier_wait(3), pthread_barrierattr_destroy(3), pthread_barrierattr_init(3)

STANDARDS

pthread_barrier_destroy() conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

pthread_barrier_init — create a barrier

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_barrier_init(pthread_barrier_t * restrict barrier,
    const pthread_barrierattr_t * restrict attr, unsigned int count);
```

DESCRIPTION

The **pthread_barrier_init**() function creates a new barrier, with attributes specified with *attr* and *count*. The *count* parameter indicates the number of threads which will participate in the barrier. If *attr* is NULL the default attributes are used. Barriers are most commonly used in the decomposition of parallel loops.

RETURN VALUES

If successful, **pthread_barrier_init**() will return zero and put the new barrier id into *barrier*, otherwise an error number will be returned to indicate the error.

ERRORS

pthread_barrier_init() shall fail if:

[EAGAIN] The system lacks the resources to initialize another barrier.

[EINVAL] The value specified by *count* is zero.

[ENOMEM] Insufficient memory exists to initialize the barrier.

pthread_barrier_init() may fail if:

[EBUSY] The barrier structure has been initialized already.

[EINVAL] The value specified by *attr* is invalid.

SEE ALSO

pthread_barrier_destroy(3), pthread_barrier_wait(3), pthread_barrierattr_destroy(3), pthread_barrierattr_init(3)

STANDARDS

pthread_barrier_init() conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

pthread_barrier_wait — wait for a barrier

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_barrier_wait(pthread_barrier_t *barrier);
```

DESCRIPTION

The **pthread_barrier_wait**() function causes the current thread to wait on the barrier specified. Once as many threads as specified by the *count* parameter to the corresponding **pthread_barrier_init**() call have called **pthread_barrier_wait**(), all threads will wake up, return from their respective **pthread_barrier_wait**() calls and continue execution.

RETURN VALUES

If successful, **pthread_barrier_wait**() will return zero for all waiting threads except for one. One thread will receive status PTHREAD_BARRIER_SERIAL_THREAD, which is intended to indicate that this thread may be used to update shared data. It is the responsibility of this thread to insure the visibility and atomicity of any updates to shared data with respect to the other threads participating in the barrier. In the case of failure, an error value will be returned.

ERRORS

pthread_barrier_wait() may fail if:

[EINVAL] The value specified by *barrier* is invalid.

SEE ALSO

pthread_barrier_destroy(3), pthread_barrier_init(3), pthread_barrierattr_destroy(3), pthread_barrierattr_init(3)

STANDARDS

pthread_barrier_wait() conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

pthread_barrierattr_init, **pthread_barrierattr_destroy**, — barrier attribute operations

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_barrierattr_init(pthread_barrierattr_t *attr);

int
pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

DESCRIPTION

Barrier attributes are used to specify parameters to **pthread_barrier_init**(). One attribute object can be used in multiple calls to **pthread_barrier_init**(), with or without modifications between calls.

The **pthread_barrierattr_init**() function initializes *attr* with all the default barrier attributes.

The **pthread_barrierattr_destroy**() function destroys *attr*.

RETURN VALUES

If successful, these functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

pthread_barrierattr_init() shall fail if:

[ENOMEM] Insufficient memory exists to initialize the barrier attributes object.

pthread_barrierattr_init() may fail if:

[EINVAL] The value specified by *attr* is invalid.

pthread_barrierattr_destroy() may fail if:

[EINVAL] The value specified by *attr* is invalid

SEE ALSO

pthread_barrier_init(3)

STANDARDS

pthread_barrierattr_init() and **pthread_barrierattr_destroy**() conform to IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

pthread_cancel — cancel execution of a thread

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_cancel(pthread_t thread);
```

DESCRIPTION

The **pthread_cancel**() function requests that *thread* be canceled. The target thread's cancelability state and type determines when the cancellation takes effect. When the cancellation is acted on, the cancellation cleanup handlers for *thread* are called. When the last cancellation cleanup handler returns, the thread-specific data destructor functions will be called for *thread*. When the last destructor function returns, *thread* will be terminated.

The cancellation processing in the target thread runs asynchronously with respect to the calling thread returning from **pthread_cancel**().

A status of PTHREAD_CANCELED is made available to any threads joining with the target. The symbolic constant PTHREAD_CANCELED expands to a constant expression of type *(void *)*, whose value matches no pointer to an object in memory nor the value NULL.

RETURN VALUES

If successful, the **pthread_cancel**() functions will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

pthread_cancel() may fail if:

[ESRCH] No thread could be found corresponding to that specified by the given thread ID.

SEE ALSO

pthread_cleanup_pop(3), pthread_cleanup_push(3), pthread_exit(3), pthread_join(3), pthread_setcancelstate(3), pthread_setcanceltype(3), pthread_testcancel(3)

STANDARDS

pthread_cancel() conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

AUTHORS

This man page was written by David Leonard ⟨d@openbsd.org⟩ for the OpenBSD implementation of **pthread_cancel**().

NAME

pthread_cleanup_push, **pthread_cleanup_pop** — add and remove cleanup functions for thread exit

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

void
pthread_cleanup_push(void (*cleanup_routine)(void *), void *arg);

void
pthread_cleanup_pop(int execute);
```

DESCRIPTION

The **pthread_cleanup_push**() function adds *cleanup_routine* to the top of the stack of cleanup handlers that get called when the current thread exits.

The **pthread_cleanup_pop**() function pops the top cleanup routine off of the current threads cleanup routine stack, and, if *execute* is non-zero, it will execute the function.

When *cleanup_routine* is called, it is passed *arg* as its only argument.

These functions may be implemented as macros which contain scope delimiters; therefore, there must be a matching **pthread_cleanup_pop**() for every **pthread_cleanup_push**() at the same level of lexical scoping.

The effect of calling **longjmp**() or **siglongjmp**() is undefined after a call to **pthread_cleanup_push**() but before the matching call to **pthread_cleanup_pop**() after the jump buffer was filled.

RETURN VALUES

Neither **pthread_cleanup_push**() nor **pthread_cleanup_pop**() returns a value.

ERRORS

None.

SEE ALSO

pthread_exit(3)

STANDARDS

pthread_cleanup_push() and **pthread_cleanup_pop**() conform to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_cond_broadcast, **pthread_cond_signal** — unblock one or all threads waiting on a condition variable

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_cond_broadcast(pthread_cond_t *cond);

int
pthread_cond_signal(pthread_cond_t *cond);
```

DESCRIPTION

The **pthread_cond_broadcast**() function unblocks all threads waiting for the condition variable *cond*. If no threads are waiting on *cond*, the **pthread_cond_broadcast**() function has no effect.

The **pthread_cond_signal**() function unblocks one thread waiting for the condition variable *cond*. If no threads are waiting on *cond*, the **pthread_cond_signal**() function has no effect.

When calling **pthread_cond_wait**() and/or **pthread_cond_timedwait**(), a temporary binding is established between the condition variable *cond* and a caller-supplied mutex.

The same mutex must be held while calling **pthread_cond_broadcast**() and **pthread_cond_signal**(). Neither function enforces this requirement, but if the mutex is not held the resulting behaviour is undefined.

RETURN VALUES

If successful, the **pthread_cond_broadcast**() and **pthread_cond_signal**() functions will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

pthread_cond_broadcast() and **pthread_cond_signal**() may fail if:

[EINVAL] The value specified by *cond* is invalid.

SEE ALSO

pthread_cond_destroy(3), pthread_cond_init(3), pthread_cond_timedwait(3), pthread_cond_wait(3)

STANDARDS

pthread_cond_broadcast() and **pthread_cond_signal**() conform to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_cond_destroy — destroy a condition variable

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_cond_destroy(pthread_cond_t *cond);
```

DESCRIPTION

The **pthread_cond_destroy**() function frees the resources allocated by the condition variable *cond*.

RETURN VALUES

If successful, the **pthread_cond_destroy**() function will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

pthread_cond_destroy() may fail if:

[EBUSY] The variable *cond* is locked by another thread.

[EINVAL] The value specified by *cond* is invalid.

SEE ALSO

pthread_cond_broadcast(3), pthread_cond_init(3), pthread_cond_signal(3), pthread_cond_timedwait(3), pthread_cond_wait(3)

STANDARDS

pthread_cond_destroy() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_cond_init — create a condition variable

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_cond_init(pthread_cond_t * restrict cond,
    const pthread_condattr_t * restrict attr);
```

DESCRIPTION

The **pthread_cond_init**() function creates a new condition variable, with attributes specified with *attr*. If *attr* is NULL the default attributes are used.

Condition variables are intended to be used to communicate changes in the state of data shared between threads. Condition variables are always associated with a mutex to provide synchronized access to the shared data. A single predicate should always be associated with a condition variable. The predicate should identify a state of the shared data that must be true before the thread proceeds.

RETURN VALUES

If successful, the **pthread_cond_init**() function will return zero and put the new condition variable id into *cond*, otherwise an error number will be returned to indicate the error.

ERRORS

pthread_cond_init() shall fail if:

[EAGAIN] The system lacks the resources to initialize another condition variable.

[ENOMEM] The process cannot allocate enough memory to initialize another condition variable.

pthread_cond_init() may fail if:

[EINVAL] The value specified by *attr* is invalid.

SEE ALSO

pthread_cond_broadcast(3), pthread_cond_destroy(3), pthread_cond_signal(3), pthread_cond_timedwait(3), pthread_cond_wait(3)

STANDARDS

pthread_cond_init() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_cond_wait, **pthread_cond_timedwait** — wait on a condition variable

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_cond_wait(pthread_cond_t * restrict cond,
    pthread_mutex_t * restrict mutex);

int
pthread_cond_timedwait(pthread_cond_t * restrict cond,
    pthread_mutex_t * restrict mutex,
    const struct timespec * restrict abstime);
```

DESCRIPTION

The **pthread_cond_wait**() function atomically blocks the current thread waiting on the condition variable specified by *cond*, and unblocks the mutex specified by *mutex*. The waiting thread unblocks after another thread calls pthread_cond_signal(3), or pthread_cond_broadcast(3) with the same condition variable. The current thread holds the lock on *mutex* upon return from the *pthread_cond_wait* call.

The **pthread_cond_timedwait**() function atomically blocks the current thread waiting on the condition variable specified by *cond*, and unblocks the mutex specified by *mutex*. The waiting thread unblocks after another thread calls pthread_cond_signal(3), or pthread_cond_broadcast(3) with the same condition variable, or if the system time reaches the time specified in *abstime*.

Note that a call to **pthread_cond_wait**() or **pthread_cond_timedwait**() may wake up spontaneously, without a call to pthread_cond_signal(3) or pthread_cond_broadcast(3). The caller should prepare for this by invoking **pthread_cond_wait**() or **pthread_cond_timedwait**() within a predicate loop that tests whether the thread should proceed.

When calling **pthread_cond_wait**() or **pthread_cond_timedwait**(), a temporary binding is established between the condition variable *cond* and the mutex *mutex*.

The same mutex must be held while calling **pthread_cond_broadcast**() and **pthread_cond_signal**() on *cond*. Additionally, the same mutex must be used for concurrent calls to **pthread_cond_wait**() and **pthread_cond_timedwait**(). Only when a condition variable is known to be quiescent may an application change the mutex associated with it. In this implementation, none of the functions enforce this requirement, but if the mutex is not held or independent mutexes are used the resulting behaviour is undefined.

RETURN VALUES

If successful, the **pthread_cond_wait**() and **pthread_cond_timedwait**() functions will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

pthread_cond_wait() may fail if:

[EINVAL] The value specified by *cond* or the value specified by *mutex* is invalid.

pthread_cond_timedwait() shall fail if:

[ETIMEDOUT] The system time has reached or exceeded the time specified in *abstime*.

pthread_cond_timedwait() may fail if:

[EINVAL] The value specified by *cond*, *mutex*, or *abstime* is invalid.

SEE ALSO

pthread_cond_broadcast(3), pthread_cond_destroy(3), pthread_cond_init(3), pthread_cond_signal(3)

STANDARDS

pthread_cond_wait() and **pthread_cond_timedwait**() conform to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

pthread_condattr_init, **pthread_condattr_destroy** — condition attribute operations

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_condattr_init(pthread_condattr_t *attr);

int
pthread_condattr_destroy(pthread_condattr_t *attr);
```

DESCRIPTION

Condition attribute objects are used to specify parameters to **pthread_cond_init**(). The **pthread_condattr_init**() function initializes a condition attribute object with the default attributes.

The **pthread_condattr_destroy**() function destroys a condition attribute object.

RETURN VALUES

If successful, these functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

pthread_condattr_init() shall fail if:

[ENOMEM] Insufficient memory exists to initialize the condition attribute object.

pthread_condattr_destroy() may fail if:

[EINVAL] The value specified by *attr* is invalid.

SEE ALSO

pthread_cond_init(3)

STANDARDS

pthread_condattr_init() and **pthread_condattr_destroy**() conform to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_create — create a new thread

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_create(pthread_t * restrict thread,
    const pthread_attr_t * restrict attr, void *(*start_routine)(void *),
    void * restrict arg);
```

DESCRIPTION

The **pthread_create**() function is used to create a new thread, with attributes specified by *attr*, within a process. If *attr* is NULL, the default attributes are used. If the attribute object pointed to by *attr* are modified later, the thread's attributes are not affected. Upon successful completion **pthread_create**() will store the ID of the created thread in the location specified by *thread*.

The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect is as if there was an implicit call to **pthread_exit**() using the return value of *start_routine* as the exit status. Note that the thread in which **main**() was originally invoked differs from this. When it returns from **main**(), the effect is as if there was an implicit call to **exit**() using the return value of **main**() as the exit status.

The signal state of the new thread is initialized as:

- The signal mask is inherited from the creating thread.
- The set of signals pending for the new thread is empty.

RETURN VALUES

If successful, the **pthread_create**() function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

pthread_create() shall fail if:

[EAGAIN] The system lacks the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process PTHREAD_THREADS_MAX would be exceeded.

[EINVAL] The value specified by *attr* is invalid.

SEE ALSO

fork(2), pthread_cleanup_pop(3), pthread_cleanup_push(3), pthread_exit(3), pthread_join(3)

STANDARDS

pthread_create() conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

pthread_detach — detach a thread

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_detach(pthread_t thread);
```

DESCRIPTION

The **pthread_detach**() function is used to indicate to the implementation that storage for the thread *thread* can be reclaimed when the thread terminates. If *thread* has not terminated, **pthread_detach**() will not cause it to terminate. The effect of multiple **pthread_detach**() calls on the same target thread is unspecified.

RETURN VALUES

If successful, the **pthread_detach**() function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

pthread_detach() shall fail if:

[EINVAL] The value specified by *thread* does not refer to a joinable thread.

[ESRCH] No thread could be found corresponding to that specified by the given thread ID, *thread*.

SEE ALSO

pthread_join(3)

STANDARDS

pthread_detach() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_equal — compare thread IDs

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_equal(pthread_t t1, pthread_t t2);
```

DESCRIPTION

The **pthread_equal**() function compares the thread IDs *t1* and *t2*.

RETURN VALUES

The **pthread_equal**() function will return non-zero if the thread IDs *t1* and *t2* correspond to the same thread, otherwise it will return zero.

ERRORS

None.

SEE ALSO

pthread_create(3), pthread_exit(3)

STANDARDS

pthread_equal() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_exit — terminate the calling thread

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

void
pthread_exit(void *value_ptr);
```

DESCRIPTION

The **pthread_exit**() function terminates the calling thread and makes the value *value_ptr* available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and are not yet popped are popped in the reverse order that they were pushed and then executed. After all cancellation handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions are called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process level cleanup actions, including, but not limited to, calling **atexit**() routines that may exist.

An implicit call to **pthread_exit**() is made when a thread other than the thread in which **main**() was first invoked returns from the start routine that was used to create it. The function's return value serves as the thread's exit status.

The behavior of **pthread_exit**() is undefined if called from a cancellation handler or destructor function that was invoked as the result of an implicit or explicit call to **pthread_exit**().

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the **pthread_exit**() *value_ptr* parameter value.

The process will exit with an exit status of 0 after the last thread has been terminated. The behavior is as if the implementation called **exit**() with a zero argument at thread termination time.

RETURN VALUES

The **pthread_exit**() function cannot return to its caller.

ERRORS

None.

SEE ALSO

_exit(2), exit(3), pthread_create(3), pthread_join(3)

STANDARDS

pthread_exit() conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

pthread_getname_np — set descriptive name of a thread

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <pthread.h>

int
pthread_getname_np(pthread_t thread, char *name, size_t len);
```

DESCRIPTION

pthread_getname_np() gets the descriptive name of the thread. It takes the following arguments.

thread The thread whose descriptive name will be obtained.

name The buffer to be filled with the descriptive name of the thread.

len The size of the buffer *name* in bytes.

RETURN VALUES

pthread_getname_np() returns 0 on success. Otherwise, **pthread_getname_np**() returns an error number described in errno(2).

SEE ALSO

errno(2), pthread_setname_np(3)

NAME

pthread_getspecific — get a thread-specific data value

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

void *
pthread_getspecific(pthread_key_t key);
```

DESCRIPTION

The **pthread_getspecific**() function returns the value currently bound to the specified *key* on behalf of the calling thread.

The effect of calling **pthread_getspecific**() with a *key* value not obtained from **pthread_key_create**() or after *key* has been deleted with **pthread_key_delete**() is undefined.

pthread_getspecific() may be called from a thread-specific data destructor function.

RETURN VALUES

The **pthread_getspecific**() function will return the thread-specific data value associated with the given *key*. If no thread-specific data value is associated with *key*, then the value NULL is returned.

ERRORS

None.

SEE ALSO

pthread_key_create(3), pthread_key_delete(3), pthread_setspecific(3)

STANDARDS

pthread_getspecific() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_join — wait for thread termination

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_join(pthread_t thread, void **value_ptr);
```

DESCRIPTION

The **pthread_join**() function suspends execution of the calling thread until the target *thread* terminates unless the target *thread* has already terminated.

On return from a successful **pthread_join**() call with a non-NULL *value_ptr* argument, the value passed to **pthread_exit**() by the terminating thread is stored in the location referenced by *value_ptr*. When a **pthread_join**() returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to **pthread_join**() specifying the same target thread are undefined. If the thread calling **pthread_join**() is cancelled, then the target thread is not detached.

A thread that has exited but remains unjoined counts against _POSIX_THREAD_THREADS_MAX.

RETURN VALUES

If successful, the **pthread_join**() function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

pthread_join() shall fail if:

[EINVAL] The value specified by *thread* does not refer to a joinable thread.

[ESRCH] No thread could be found corresponding to that specified by the given thread ID, *thread*.

pthread_join() may fail if:

[EDEADLK] A deadlock was detected or the value of *thread* specifies the calling thread.

SEE ALSO

wait(2), pthread_create(3)

STANDARDS

pthread_join() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_key_create — thread-specific data key creation

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
```

DESCRIPTION

The **pthread_key_create**() function creates a thread-specific data key visible to all threads in the process. Key values provided by **pthread_key_create**() are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by **pthread_setspecific**() are maintained on a per-thread basis and persist for the life of the calling thread.

Upon key creation, the value NULL is associated with the new key in all active threads. Upon thread creation, the value NULL is associated with all defined keys in the new thread.

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with the key, the function pointed to is called with the current associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

If, after all the destructors have been called for all non-NULL values with associated destructors, there are still some non-NULL values with associated destructors, then the process is repeated. If, after at least PTHREAD_DESTRUCTOR_ITERATIONS iterations of destructor calls for outstanding non-NULL values, there are still some non-NULL values with associated destructors, the implementation stops calling destructors.

RETURN VALUES

If successful, the **pthread_key_create**() function will store the newly created key value at the location specified by *key* and returns zero. Otherwise an error number will be returned to indicate the error.

ERRORS

pthread_key_create() shall fail if:

[EAGAIN] The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per process PTHREAD_KEYS_MAX would be exceeded.

[ENOMEM] Insufficient memory exists to create the key.

SEE ALSO

pthread_getspecific(3), pthread_key_delete(3), pthread_setspecific(3)

STANDARDS

pthread_key_create() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_key_delete — delete a thread-specific data key

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_key_delete(pthread_key_t key);
```

DESCRIPTION

The **pthread_key_delete**() function deletes a thread-specific data key previously returned by **pthread_key_create**(). The thread-specific data values associated with *key* need not be NULL at the time that **pthread_key_delete**() is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after **pthread_key_delete**() is called. Any attempt to use *key* following the call to **pthread_key_delete**() results in undefined behavior.

The **pthread_key_delete**() function is callable from within destructor functions. Destructor functions are not invoked by **pthread_key_delete**(). Any destructor function that may have been associated with *key* will no longer be called upon thread exit.

RETURN VALUES

If successful, the **pthread_key_delete**() function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

pthread_key_delete() may fail if:

[EINVAL] The *key* value is invalid.

SEE ALSO

pthread_getspecific(3), pthread_key_create(3), pthread_setspecific(3)

STANDARDS

pthread_key_delete() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

BUGS

The current specifications of **pthread_key_create**() and **pthread_key_delete**() are flawed and do not permit a clean implementation without potential problems. The current implementation of these functions NetBSD in addresses these problems by not supporting key reuse.

NAME

pthread_kill — send a signal to a specified thread

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>
#include <signal.h>

int
pthread_kill(pthread_t thread, int sig);
```

DESCRIPTION

The **pthread_kill**() function sends a signal, specified by *sig*, to a thread, specified by *thread*. The signal will be handled in the context of *thread*, but the signal action may alter the process as a whole. If *sig* is 0, error checking is performed, but no signal is actually sent.

RETURN VALUES

If successful, **pthread_kill**() returns 0. Otherwise, an error number is returned.

ERRORS

pthread_kill() shall fail if:

[EINVAL] *sig* is an invalid or unsupported signal number.

[ESRCH] *thread* is an invalid thread ID.

SEE ALSO

kill(2), sigwait(2), pthread_self(3), raise(3)

STANDARDS

pthread_kill() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_mutex_destroy — free resources allocated for a mutex

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutex_destroy(pthread_mutex_t *mutex);
```

DESCRIPTION

The **pthread_mutex_destroy**() function frees the resources allocated for *mutex*.

RETURN VALUES

If successful, **pthread_mutex_destroy**() will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

pthread_mutex_destroy() may fail if:

[EBUSY] *Mutex* is locked by another thread.

[EINVAL] The value specified by *mutex* is invalid.

SEE ALSO

pthread_mutex_init(3), pthread_mutex_lock(3), pthread_mutex_trylock(3), pthread_mutex_unlock(3)

STANDARDS

pthread_mutex_destroy() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_mutex_init — create a mutex

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutex_init(pthread_mutex_t * restrict mutex,
    const pthread_mutexattr_t * restrict attr);
```

DESCRIPTION

The **pthread_mutex_init**() function creates a new mutex, with attributes specified with *attr*. If *attr* is NULL the default attributes are used.

RETURN VALUES

If successful, **pthread_mutex_init**() will return zero and put the new mutex id into *mutex*, otherwise an error number will be returned to indicate the error.

ERRORS

pthread_mutex_init() shall fail if:

[EAGAIN] The system lacks the resources to initialize another mutex.

[ENOMEM] The process cannot allocate enough memory to initialize another mutex.

pthread_mutex_init() may fail if:

[EINVAL] The value specified by *attr* is invalid.

SEE ALSO

pthread_mutex_destroy(3), pthread_mutex_lock(3), pthread_mutex_trylock(3), pthread_mutex_unlock(3)

STANDARDS

pthread_mutex_init() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_mutex_lock, **pthread_mutex_trylock** — acquire a lock on a mutex

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutex_lock(pthread_mutex_t *mutex);

int
pthread_mutex_trylock(pthread_mutex_t *mutex);
```

DESCRIPTION

The **pthread_mutex_lock**() function locks *mutex*. If the mutex is already locked, the calling thread will block until the mutex becomes available.

The **pthread_mutex_trylock**() function locks *mutex*. If the mutex is already locked, **pthread_mutex_trylock**() will not block waiting for the mutex, but will return an error condition.

RETURN VALUES

If successful, **pthread_mutex_lock**() and **pthread_mutex_trylock**() will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

pthread_mutex_lock() may fail if:

[EDEADLK] A deadlock would occur if the thread blocked waiting for *mutex*.

[EINVAL] The value specified by *mutex* is invalid.

pthread_mutex_trylock() shall fail if:

[EBUSY] *Mutex* is already locked.

pthread_mutex_trylock() may fail if:

[EINVAL] The value specified by *mutex* is invalid.

SEE ALSO

pthread_mutex_destroy(3), pthread_mutex_init(3), pthread_mutex_unlock(3)

STANDARDS

pthread_mutex_lock() and **pthread_mutex_trylock**() conform to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_mutex_unlock — unlock a mutex

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

If the current thread holds the lock on *mutex*, then the **pthread_mutex_unlock**() function unlocks *mutex*.

RETURN VALUES

If successful, **pthread_mutex_unlock**() will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

pthread_mutex_unlock() may fail if:

[EINVAL] The value specified by *mutex* is invalid.

[EPERM] The current thread does not hold a lock on *mutex*.

SEE ALSO

pthread_mutex_destroy(3), pthread_mutex_init(3), pthread_mutex_lock(3), pthread_mutex_trylock(3)

STANDARDS

pthread_mutex_unlock() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_mutexattr_init, **pthread_mutexattr_destroy**, **pthread_mutexattr_settype**, **pthread_mutexattr_gettype** — mutex attribute operations

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutexattr_init(pthread_mutexattr_t *attr);

int
pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

int
pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);

int
pthread_mutexattr_gettype(pthread_mutexattr_t * restrict attr,
    int * restrict type);
```

DESCRIPTION

Mutex attributes are used to specify parameters to **pthread_mutex_init**(). One attribute object can be used in multiple calls to **pthread_mutex_init**(), with or without modifications between calls.

The **pthread_mutexattr_init**() function initializes *attr* with all the default mutex attributes.

The **pthread_mutexattr_destroy**() function destroys *attr*.

The **pthread_mutexattr_settype**() functions set the mutex type value of the attribute. Valid mutex types are: PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_ERRORCHECK, PTHREAD_MUTEX_RECURSIVE, and PTHREAD_MUTEX_DEFAULT. The default mutex type for **pthread_mutexaddr_init**() is PTHREAD_MUTEX_DEFAULT.

PTHREAD_MUTEX_NORMAL mutexes do not check for usage errors. PTHREAD_MUTEX_NORMAL mutexes will deadlock if reentered, and result in undefined behavior if a locked mutex is unlocked by another thread. Attempts to unlock an already unlocked PTHREAD_MUTEX_NORMAL mutex will result in undefined behavior.

PTHREAD_MUTEX_ERRORCHECK mutexes do check for usage errors. If an attempt is made to relock a PTHREAD_MUTEX_ERRORCHECK mutex without first dropping the lock an error will be returned. If a thread attempts to unlock a PTHREAD_MUTEX_ERRORCHECK mutex that is locked by another thread, an error will be returned. If a thread attempts to unlock a PTHREAD_MUTEX_ERRORCHECK thread that is unlocked, an error will be returned.

PTHREAD_MUTEX_RECURSIVE mutexes allow recursive locking. An attempt to relock a PTHREAD_MUTEX_RECURSIVE mutex that is already locked by the same thread succeeds. An equivalent number of pthread_mutex_unlock(3) calls are needed before the mutex will wake another thread waiting on this lock. If a thread attempts to unlock a PTHREAD_MUTEX_RECURSIVE mutex that is locked by another thread, an error will be returned. If a thread attempts to unlock a PTHREAD_MUTEX_RECURSIVE thread that is unlocked, an error will be returned.

PTHREAD_MUTEX_DEFAULT mutexes result in undefined behavior if reentered. Unlocking a PTHREAD_MUTEX_DEFAULT mutex locked by another thread will result in undefined behavior. Attempts to unlock an already unlocked PTHREAD_MUTEX_DEFAULT mutex will result in undefined behavior.

pthread_mutexattr_gettype() functions copy the type value of the attribute to the location pointed to by the second parameter.

RETURN VALUES

If successful, these functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

pthread_mutexattr_init() shall fail if:

[ENOMEM] Insufficient memory exists to initialize the mutex attributes object.

pthread_mutexattr_settype() shall fail if:

[EINVAL] The value specified by *type* is invalid.

pthread_mutexattr_destroy(), **pthread_mutexattr_settype**(), and **pthread_mutexattr_gettype**() may fail if:

[EINVAL] Invalid value for *attr*.

SEE ALSO

pthread_mutex_init(3)

STANDARDS

pthread_mutexattr_init(), **pthread_mutexattr_destroy**(), **pthread_mutexattr_settype**(), and **pthread_mutexattr_gettype**() conform to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

pthread_once — dynamic package initialization

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int
pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

DESCRIPTION

The first call to **pthread_once**() by any thread in a process, with a given *once_control*, will call the **init_routine**() with no arguments. Subsequent calls to **pthread_once**() with the same *once_control* will not call the **init_routine**(). On return from **pthread_once**(), it is guaranteed that **init_routine**() has completed. The *once_control* parameter is used to determine whether the associated initialization routine has been called.

The function **pthread_once**() is not a cancellation point. However, if **init_routine**() is a cancellation point and is cancelled, the effect on *once_control* is as if **pthread_once**() was never called.

The constant *PTHREAD_ONCE_INIT* is defined by header ⟨pthread.h⟩.

The behavior of **pthread_once**() is undefined if *once_control* has automatic storage duration or is not initialized by *PTHREAD_ONCE_INIT*.

RETURN VALUES

If successful, the **pthread_once**() function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

None.

STANDARDS

pthread_once() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_rwlock_destroy — destroy a read/write lock

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlock_destroy(pthread_rwlock_t *lock);
```

DESCRIPTION

The **pthread_rwlock_destroy**() function is used to destroy a read/write lock previously created with **pthread_rwlock_init**().

RETURN VALUES

If successful, the **pthread_rwlock_destroy**() function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlock_destroy**() function may fail if:

[EBUSY] The system has detected an attempt to destroy the object referenced by *lock* while it is locked.

[EINVAL] The value specified by *lock* is invalid.

SEE ALSO

pthread_rwlock_init(3), pthread_rwlock_rdlock(3), pthread_rwlock_unlock(3), pthread_rwlock_wrlock(3)

STANDARDS

pthread_rwlock_destroy() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_rwlock_init — initialize a read/write lock

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlock_init(pthread_rwlock_t * restrict lock,
    const pthread_rwlockattr_t * restrict attr);
```

DESCRIPTION

The **pthread_rwlock_init**() function is used to initialize a read/write lock, with attributes specified by *attr*. If *attr* is NULL, the default read/write lock attributes are used.

The results of calling **pthread_rwlock_init**() with an already initialized lock are undefined.

RETURN VALUES

If successful, the **pthread_rwlock_init**() function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlock_init**() function shall fail if:

[EAGAIN] The system lacks the resources to initialize another read-write lock.

[ENOMEM] Insufficient memory exists to initialize the read-write lock.

The **pthread_rwlock_init**() function may fail if:

[EBUSY] The system has detected an attempt to re-initialize the object referenced by *lock*, a previously initialized but not yet destroyed read/write lock.

[EINVAL] The value specified by *attr* is invalid.

SEE ALSO

pthread_rwlock_destroy(3), pthread_rwlock_rdlock(3), pthread_rwlock_unlock(3), pthread_rwlock_wrlock(3)

STANDARDS

pthread_rwlock_init() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

BUGS

The PTHREAD_PROCESS_SHARED attribute is not supported.

NAME

pthread_rwlock_rdlock, **pthread_rwlock_timedrdlock**, **pthread_rwlock_tryrdlock** — acquire a read/write lock for reading

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlock_rdlock(pthread_rwlock_t *lock);

int
pthread_rwlock_timedrdlock(pthread_rwlock_t * restrict lock,
    const struct timespec * restrict abstime);

int
pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
```

DESCRIPTION

The **pthread_rwlock_rdlock**() function acquires a read lock on *lock* provided that *lock* is not presently held for writing and no writer threads are presently blocked on the lock. If the read lock cannot be immediately acquired, the calling thread blocks until it can acquire the lock.

The **pthread_rwlock_timedrdlock**() performs the same action, but will not wait beyond *abstime* to obtain the lock before returning.

The **pthread_rwlock_tryrdlock**() function performs the same action as **pthread_rwlock_rdlock**(), but does not block if the lock cannot be immediately obtained (i.e., the lock is held for writing or there are waiting writers).

A thread may hold multiple concurrent read locks. If so, **pthread_rwlock_unlock**() must be called once for each lock obtained.

The results of acquiring a read lock while the calling thread holds a write lock are undefined.

RETURN VALUES

If successful, the **pthread_rwlock_rdlock**(), **pthread_rwlock_timedrdlock**(), and **pthread_rwlock_tryrdlock**() functions will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlock_tryrdlock**() function shall fail if:

[EBUSY] The lock could not be acquired because a writer holds the lock or was blocked on it.

The **pthread_rwlock_timedrdlock**() function shall fail if:

[ETIMEDOUT] The time specified by *abstime* was reached before the lock could be obtained.

The **pthread_rwlock_rdlock**(), **pthread_rwlock_timedrdlock**(), and **pthread_rwlock_tryrdlock**() functions may fail if:

[EAGAIN] The lock could not be acquired because the maximum number of read locks against *lock* has been exceeded.

[EDEADLK] The current thread already owns *lock* for writing.

[EINVAL] The value specified by *lock* is invalid.

SEE ALSO

`pthread_rwlock_destroy`(3), `pthread_rwlock_init`(3), `pthread_rwlock_unlock`(3), `pthread_rwlock_wrlock`(3)

STANDARDS

`pthread_rwlock_rdlock`(), **`pthread_rwlock_timedrdlock`**(), and **`pthread_rwlock_tryrdlock`**() conform to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

pthread_rwlock_unlock — release a read/write lock

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

DESCRIPTION

The **pthread_rwlock_unlock**() function is used to release the read/write lock previously obtained by **pthread_rwlock_rdlock**(), **pthread_rwlock_wrlock**(), **pthread_rwlock_tryrdlock**(), or **pthread_rwlock_trywrlock**().

RETURN VALUES

If successful, the **pthread_rwlock_unlock**() function will return zero. Otherwise an error number will be returned to indicate the error.

The results are undefined if *lock* is not held by the calling thread.

ERRORS

The **pthread_rwlock_unlock**() function may fail if:

[EINVAL]	The value specified by *lock* is invalid.
[EPERM]	The current thread does not own the read/write lock.

SEE ALSO

pthread_rwlock_destroy(3), pthread_rwlock_init(3), pthread_rwlock_rdlock(3), pthread_rwlock_wrlock(3)

STANDARDS

pthread_rwlock_unlock() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_rwlock_wrlock, **pthread_rwlock_timedwrlock**, **pthread_rwlock_trywrlock** — acquire a read/write lock for writing

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlock_wrlock(pthread_rwlock_t *lock);

int
pthread_rwlock_timedwrlock(pthread_rwlock_t * restrict lock,
    const struct timespec * restrict abstime);

int
pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
```

DESCRIPTION

The **pthread_rwlock_wrlock**() function blocks until a write lock can be acquired against *lock*.

The **pthread_rwlock_timedwrlock**() performs the same action, but will not wait beyond *abstime* to obtain the lock before returning.

The **pthread_rwlock_trywrlock**() function performs the same action as **pthread_rwlock_wrlock**(), but does not block if the lock cannot be immediately obtained.

The results are undefined if the calling thread already holds the lock at the time the call is made.

RETURN VALUES

If successful, the **pthread_rwlock_wrlock**(), **pthread_rwlock_timedwrlock**(), and **pthread_rwlock_trywrlock**() functions will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlock_trywrlock**() function shall fail if:

[EBUSY] The calling thread is not able to acquire the lock without blocking.

The **pthread_rwlock_timedrdlock**() function shall fail if:

[ETIMEDOUT] The time specified by *abstime* was reached before the lock could be obtained.

The **pthread_rwlock_wrlock**(), **pthread_rwlock_timedwrlock**(), and **pthread_rwlock_trywrlock**() functions may fail if:

[EDEADLK] The calling thread already owns the read/write lock (for reading or writing).

[EINVAL] The value specified by *lock* is invalid.

SEE ALSO

pthread_rwlock_destroy(3), pthread_rwlock_init(3), pthread_rwlock_rdlock(3), pthread_rwlock_unlock(3)

STANDARDS

pthread_rwlock_wrlock(), **pthread_rwlock_timedwrlock**(), and **pthread_rwlock_trywrlock**() conform to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_rwlockattr_init, **pthread_rwlockattr_destroy** — initialize or destroy read/write lock attributes

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

int
pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

DESCRIPTION

The **pthread_rwlockattr_init**() function is used to initialize a read/write lock attributes object.

The **pthread_rwlockattr_destroy**() function is used to destroy a read/write lock attribute object previously created with **pthread_rwlockattr_init**().

RETURN VALUES

If successful, the **pthread_rwlockattr_init**() and **pthread_rwlockattr_destroy**() functions return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

pthread_rwlockattr_init() shall fail if:

[ENOMEM] Insufficient memory exists to initialize the read/write lock attributes object.

pthread_rwlockattr_init() and **pthread_rwlockattr_destroy**() may fail if:

[EINVAL] The value specified by *attr* is invalid.

SEE ALSO

pthread_rwlock_init(3)

STANDARDS

The **pthread_rwlockattr_init**() and **pthread_rwlockattr_destroy**() functions conform to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_setschedparam, **pthread_getschedparam** — thread scheduling parameter manipulation

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_setschedparam(pthread_t thread, int policy,
    const struct sched_param *param);

int
pthread_getschedparam(pthread_t thread, int * restrict policy,
    struct sched_param * restrict param);
```

DESCRIPTION

The **pthread_setschedparam**() and **pthread_getschedparam**() functions set and get the scheduling parameters of individual threads. The scheduling policy for a thread can either be SCHED_FIFO (first in, first out), SCHED_RR (round-robin), or SCHED_OTHER (system default). The thread priority (accessed via *param->sched_priority*) must be at least PTHREAD_MIN_PRIORITY and no more than PTHREAD_MAX_PRIORITY.

RETURN VALUES

If successful, these functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

pthread_setschedparam() may fail if:

[EINVAL] The value specified by *policy* is invalid.

[ENOTSUP] Invalid value for scheduling parameters.

[ESRCH] Non-existent thread *thread*.

pthread_getschedparam() may fail if:

[ESRCH] Non-existent thread *thread*.

STANDARDS

pthread_setschedparam() and **pthread_getschedparam**() conform to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_self — get the calling thread's ID

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

pthread_t
pthread_self(void);
```

DESCRIPTION

The **pthread_self**() function returns the thread ID of the calling thread.

RETURN VALUES

The **pthread_self**() function returns the thread ID of the calling thread.

ERRORS

None.

SEE ALSO

pthread_create(3), pthread_equal(3)

STANDARDS

pthread_self() conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

pthread_setname_np — set descriptive name of a thread

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <pthread.h>

int
pthread_setname_np(pthread_t thread, const char *name, void *arg);
```

DESCRIPTION

pthread_setname_np() sets the descriptive name of the thread. It takes the following arguments.

thread The thread whose descriptive name will be set.

name The `printf`(3) format string to be used to construct the descriptive name of the thread. The resulted descriptive name should be shorter than `PTHREAD_MAX_NAMELEN_NP`.

arg The `printf`(3) argument used with *name*.

RETURN VALUES

pthread_setname_np() returns 0 on success. Otherwise, **pthread_setname_np**() returns an error number described in `errno`(2).

SEE ALSO

`errno`(2), `pthread_getname_np`(3)

NAME

pthread_setspecific — set a thread-specific data value

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_setspecific(pthread_key_t key, const void *value);
```

DESCRIPTION

The **pthread_setspecific**() function associates a thread-specific value with a *key* obtained via a previous call to **pthread_key_create**(). Different threads have different values bound to each key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The effect of calling **pthread_setspecific**() with a key value not obtained from **pthread_key_create**() or after *key* has been deleted with **pthread_key_delete**() is undefined.

pthread_setspecific() may be called from a thread-specific data destructor function, however this may result in lost storage or infinite loops.

RETURN VALUES

If successful, the **pthread_setspecific**() function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

pthread_setspecific() shall fail if:

[ENOMEM] Insufficient memory exists to associate the value with the *key*.

pthread_setspecific() may fail if:

[EINVAL] The *key* value is invalid.

SEE ALSO

pthread_getspecific(3), pthread_key_create(3), pthread_key_delete(3)

STANDARDS

pthread_setspecific() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

pthread_sigmask — examine and/or change a thread's signal mask

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <signal.h>

int
pthread_sigmask(int how, const sigset_t * restrict set,
    sigset_t * restrict oset);
```

DESCRIPTION

The **pthread_sigmask**() function examines and/or changes the calling thread's signal mask.

If *set* is not NULL, it specifies a set of signals to be modified, and *how* specifies what to set the signal mask to:

SIG_BLOCK Union of the current mask and *set*.

SIG_UNBLOCK Intersection of the current mask and the complement of *set*.

SIG_SETMASK *set*.

If *oset* is not NULL, the previous signal mask is stored in the location pointed to by *oset*.

SIGKILL and SIGSTOP cannot be blocked, and will be silently ignored if included in the signal mask.

RETURN VALUES

If successful, **pthread_sigmask**() returns 0. Otherwise, an error is returned.

ERRORS

pthread_sigmask() shall fail if:

[EINVAL] *how* is not one of the defined values.

SEE ALSO

sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2), sigwait(2), sigsetops(3)

STANDARDS

pthread_sigmask() conforms to ISO/IEC 9945-1:1996 ("POSIX.1")

NAME

pthread_spin_destroy — destroy a spin lock

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_spin_destroy(pthread_spinlock_t *lock);
```

DESCRIPTION

The **pthread_spin_destroy**() function is used to destroy a spin lock previously created with **pthread_spin_init**().

RETURN VALUES

If successful, the **pthread_spin_destroy**() function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_spin_destroy**() function may fail if:

[EBUSY] The system has detected an attempt to destroy the object referenced by *lock* while it is locked.

[EINVAL] The value specified by *lock* is invalid.

SEE ALSO

pthread_spin_init(3), pthread_spin_lock(3), pthread_spin_trylock(3), pthread_spin_unlock(3)

STANDARDS

pthread_spin_destroy() conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

CAVEATS

Applications using spinlocks are vulnerable to the effects of priority inversion. Applications using real-time threads (SCHED_FIFO, SCHED_RR) should not use these interfaces. Outside carefully controlled environments, priority inversion with spinlocks can lead to system deadlock. Mutexes are preferable in nearly every possible use case.

NAME

pthread_spin_init — initialize a spin lock

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

DESCRIPTION

The **pthread_spin_init**() function is used to initialize a spinlock. The *pshared* parameter is currently unused and all spinlocks exhibit the PTHREAD_PROCESS_SHARED property.

The results of calling **pthread_spin_init**() with an already initialized lock are undefined.

RETURN VALUES

If successful, the **pthread_spin_init**() function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_spin_init**() function shall fail if:

[ENOMEM] Insufficient memory exists to initialize the lock.

The **pthread_spin_init**() function may fail if:

[EINVAL] The *lock* parameter was NULL or the *pshared* parameter was neither PTHREAD_PROCESS_SHARED nor PTHREAD_PROCESS_PRIVATE.

SEE ALSO

pthread_spin_destroy(3), pthread_spin_lock(3), pthread_spin_trylock(3), pthread_spin_unlock(3)

STANDARDS

pthread_spin_init() conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

CAVEATS

Applications using spinlocks are vulnerable to the effects of priority inversion. Applications using real-time threads (SCHED_FIFO, SCHED_RR) should not use these interfaces. Outside carefully controlled environments, priority inversion with spinlocks can lead to system deadlock. Mutexes are preferable in nearly every possible use case.

NAME

pthread_spin_lock, **pthread_spin_trylock** — acquire a spin lock

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_spin_lock(pthread_spinlock_t *lock);

int
pthread_spin_trylock(pthread_spinlock_t *lock);
```

DESCRIPTION

The **pthread_spin_lock**() function acquires a spin lock on *lock* provided that *lock* is not presently held. If the lock cannot be immediately acquired, the calling thread repeatedly retries until it can acquire the lock.

The **pthread_spin_trylock**() function performs the same action, but does not block if the lock cannot be immediately obtained (i.e., the lock is held).

RETURN VALUES

If successful, the **pthread_spin_lock**() and **pthread_spin_trylock**() functions will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_spin_trylock**() function shall fail if:

[EBUSY] The lock could not be acquired because a writer holds the lock or was blocked on it.

The **pthread_spin_lock**() function may fail if:

[EDEADLK] The current thread already owns *lock* for writing.

The **pthread_spin_lock**() and **pthread_spin_trylock**() functions may fail if:

[EINVAL] The value specified by *lock* is invalid.

SEE ALSO

pthread_spin_destroy(3), pthread_spin_init(3), pthread_spin_unlock(3)

STANDARDS

pthread_spin_lock() and **pthread_spin_trylock**() conform to IEEE Std 1003.1-2001 (“POSIX.1”).

CAVEATS

Applications using spinlocks are vulnerable to the effects of priority inversion. Applications using real-time threads (SCHED_FIFO, SCHED_RR) should not use these interfaces. Outside carefully controlled environments, priority inversion with spinlocks can lead to system deadlock. Mutexes are preferable in nearly every possible use case.

NAME

pthread_spin_unlock — release a spin lock

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_spin_unlock(pthread_spinlock_t *lock);
```

DESCRIPTION

The **pthread_spin_unlock**() function is used to release the read/write lock previously obtained by **pthread_spin_lock**() or **pthread_spin_trylock**().

RETURN VALUES

If successful, the **pthread_spin_unlock**() function will return zero. Otherwise an error number will be returned to indicate the error.

The results are undefined if *lock* is not held by the calling thread.

ERRORS

The **pthread_spin_unlock**() function may fail if:

[EINVAL] The value specified by *lock* is invalid.

SEE ALSO

pthread_spin_destroy(3), pthread_spin_init(3), pthread_spin_lock(3), pthread_spin_trylock(3)

STANDARDS

pthread_rwlock_unlock() conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

CAVEATS

Applications using spinlocks are vulnerable to the effects of priority inversion. Applications using real-time threads (SCHED_FIFO, SCHED_RR) should not use these interfaces. Outside carefully controlled environments, priority inversion with spinlocks can lead to system deadlock. Mutexes are preferable in nearly every possible use case.

NAME

pthread_suspend_np, **pthread_resume_np** — suspend/resume the given thread

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_suspend_np(pthread_t thread);

int
pthread_resume_np(pthread_t thread);
```

DESCRIPTION

The **pthread_suspend_np**() function suspends the *thread* given as argument. If *thread* is the currently running thread as returned by pthread_self(3), the function fails and returns EDEADLK. Otherwise, it removes the named thread from the running queue, and adds it to the suspended queue. The *thread* will remain blocked until **pthread_resume_np**() is called on it.

pthread_resume_np() resumes the *thread* given as argument, if it was suspended.

RETURN VALUES

The **pthread_suspend_np**() function returns 0 on success and an error number indicating the reason for the failure. The **pthread_resume_np**() function always returns 0.

ERRORS

pthread_suspend_np() shall fail if:

[EDEADLK] The thread requested to suspend was the currently running thread.

pthread_resume_np() never fails.

NOTES

Some **pthread_suspend_np**() implementations may allow suspending the current thread. This is dangerous, because the semantics of the function would then require the scheduler to schedule another thread, causing a thread context switch. Since that context switch can happen in a signal handler by someone calling **pthread_suspend_np**() in a signal handler, this is currently not allowed.

In **pthread_resume_np**() we don't check if the *thread* argument is not already suspended. Some implementations might return an error condition if **pthread_resume_np**() is called on a non-suspended thread.

SEE ALSO

pthread_attr_setcreatesuspend_np(3), pthread_self(3)

NAME

pthread_setcancelstate, **pthread_setcanceltype**, **pthread_testcancel** — set cancelability state

LIBRARY

POSIX Threads Library (libpthread, −lpthread)

SYNOPSIS

```
#include <pthread.h>

int
pthread_setcancelstate(int state, int *oldstate);

int
pthread_setcanceltype(int type, int *oldtype);

void
pthread_testcancel(void);
```

DESCRIPTION

The **pthread_setcancelstate**() function atomically both sets the calling thread's cancelability state to the indicated *state* and, if *oldstate* is not NULL, returns the previous cancelability state at the location referenced by *oldstate*. Legal values for *state* are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DISABLE.

The **pthread_setcanceltype**() function atomically both sets the calling thread's cancelability type to the indicated *type* and, if *oldtype* is not NULL, returns the previous cancelability type at the location referenced by *oldtype*. Legal values for *type* are PTHREAD_CANCEL_DEFERRED and PTHREAD_CANCEL_ASYNCHRONOUS.

The cancelability state and type of any newly created threads, including the thread in which **main**() was first invoked, are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DEFERRED respectively.

The **pthread_testcancel**() function creates a cancellation point in the calling thread. The **pthread_testcancel**() function has no effect if cancelability is disabled.

Cancelability States

The cancelability state of a thread determines the action taken upon receipt of a cancellation request. The thread may control cancellation in a number of ways.

Each thread maintains its own "cancelability state" which may be encoded in two bits:

Cancelability Enable When cancelability is PTHREAD_CANCEL_DISABLE, cancellation requests against the target thread are held pending.

Cancelability Type When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_ASYNCHRONOUS, new or pending cancellation requests may be acted upon at any time. When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_DEFERRED, cancellation requests are held pending until a cancellation point (see below) is reached. If cancelability is disabled, the setting of the cancelability type has no immediate effect as all cancellation requests are held pending; however, once cancelability is enabled again the new type will be in effect.

Cancellation Points

Cancellation points will occur when a thread is executing the following functions: **accept**(), **close**(), **connect**(), **creat**(), **fcntl**(), **fsync**(), **fsync_range**(), **msgrcv**(), **msgsnd**(), **msync**(),

nanosleep(), **open**(), **pause**(), **poll**(), **pread**(), **pselect**(), **pthread_cond_timedwait**(), **pthread_cond_wait**(), **pthread_join**(), **pthread_testcancel**(), **pwrite**(), **read**(), **readv**(), **recv**(), **recvfrom**(), **recvmsg**(), **select**(), **sem_timedwait**(), **sem_wait**(), **send**(), **sendmsg**(), **sendto**(), **sigpause**(), **sigsuspend**(), **sigtimedwait**(), **sigwait**(), **sigwaitinfo**(), **sleep**(), **system**(), **tcdrain**(), **usleep**(), **wait**(), **waitid**(), **waitpid**(), **write**(), and **writev**().

RETURN VALUES

If successful, the **pthread_setcancelstate**() and **pthread_setcanceltype**() functions will return zero. Otherwise, an error number shall be returned to indicate the error.

The **pthread_setcancelstate**() and **pthread_setcanceltype**() functions are used to control the points at which a thread may be asynchronously canceled. For cancellation control to be usable in modular fashion, some rules must be followed.

For purposes of this discussion, consider an object to be a generalization of a procedure. It is a set of procedures and global variables written as a unit and called by clients not known by the object. Objects may depend on other objects.

First, cancelability should only be disabled on entry to an object, never explicitly enabled. On exit from an object, the cancelability state should always be restored to its value on entry to the object.

This follows from a modularity argument: if the client of an object (or the client of an object that uses that object) has disabled cancelability, it is because the client doesn't want to have to worry about how to clean up if the thread is canceled while executing some sequence of actions. If an object is called in such a state and it enables cancelability and a cancellation request is pending for that thread, then the thread will be canceled, contrary to the wish of the client that disabled.

Second, the cancelability type may be explicitly set to either *deferred* or *asynchronous* upon entry to an object. But as with the cancelability state, on exit from an object that cancelability type should always be restored to its value on entry to the object.

Finally, only functions that are cancel-safe may be called from a thread that is asynchronously cancelable.

ERRORS

The function **pthread_setcancelstate**() may fail with:

[EINVAL] The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

The function **pthread_setcanceltype**() may fail with:

[EINVAL] The specified state is not PTHREAD_CANCEL_DEFERRED or PTHREAD_CANCEL_ASYNCHRONOUS.

SEE ALSO

pthread_cancel(3)

STANDARDS

pthread_testcancel() conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

AUTHORS

This man page was written by David Leonard ⟨d@openbsd.org⟩ for the OpenBSD implementation of pthread_cancel(3).

NAME

ptsname — get the pathname of the slave pseudo-terminal device

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

char *
ptsname(int masterfd);
```

DESCRIPTION

The **ptsname**() function returns the pathname of the slave pseudo-terminal device that corresponds to the master pseudo-terminal device associated with *masterfd*. The **ptsname**() function is not reentrant or thread-safe.

RETURN VALUES

If successful, **ptsname**() returns a pointer to a nul-terminated string containing the pathname of the slave pseudo-terminal device. If an error occurs **ptsname**() will return NULL and *errno* is set to indicate the error.

ERRORS

The **ptsname**() function will fail if:

[EACCESS] the corresponding pseudo-terminal device could not be accessed.

[EBADF] *masterfd* is not a valid descriptor.

[EINVAL] *masterfd* is not associated with a master pseudo-terminal device.

NOTES

The error returns of **ptsname**() are a NetBSD extension. The **ptsname**() function is equivalent to:

```
struct ptmget pm;
return ioctl(masterfd, TIOCPTSNAME, &pm) == -1 ? NULL : pm.ps;
```

SEE ALSO

ioctl(2), grantpt(3), posix_openpt(3), unlockpt(3)

STANDARDS

The **ptsname**() function conforms to IEEE Std 1003.1-2001 (“POSIX.1”). Its first release was in X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”).

NAME

puffs — Pass-to-Userspace Framework File System development interface

LIBRARY

library “libpuffs”

SYNOPSIS

```
#include <puffs.h>

struct puffs_usermount *
puffs_init(struct puffs_ops *pops, const char *mntfromname,
    const char *puffsname, void *private, uint32_t flags);

int
puffs_mount(struct puffs_usermount *pu, const char *dir, int mntflags,
    void *root_cookie);

int
puffs_getselectable(struct puffs_usermount *pu);

int
puffs_setblockingmode(struct puffs_usermount *pu, int mode);

int
puffs_getstate(struct puffs_usermount *pu);

int
puffs_setstacksize(struct puffs_usermount *pu, size_t stacksize);

void
puffs_setroot(struct puffs_usermount *pu, struct puffs_node *node);

void
puffs_setrootinfo(struct puffs_usermount *pu, enum vtype vt, vsize_t vsize,
    dev_t rdev);

struct puffs_node *
puffs_getroot(struct puffs_usermount *pu);

void *
puffs_getspecific(struct puffs_usermount *pu);

void
puffs_setmaxreqlen(struct puffs_usermount *pu, size_t maxreqlen);

size_t
puffs_getmaxreqlen(struct puffs_usermount *pu);

void
puffs_setfhsize(struct puffs_usermount *pu, size_t fhsize, int flags);

void
puffs_setncookiehash(struct puffs_usermount *pu, int nhashes);

void
puffs_ml_loop_fn(struct puffs_usermount *pu);

void
puffs_ml_setloopfn(struct puffs_usermount *pu, puffs_ml_loop_fn lfn);
```

void
puffs_ml_settimeout(*struct puffs_usermount *pu*, *struct timespec *ts*);

int
puffs_daemon(*struct puffs_usermount *pu*, *int nochdir*, *int noclose*);

int
puffs_mainloop(*struct puffs_usermount *pu*);

int
puffs_dispatch_create(*struct puffs_usermount *pu*,
*struct puffs_framebuf *pb*,*struct puffs_cc **pccp*);

int
puffs_dispatch_exec(*.Fa*, *struct puffs_cc *pcc*, *struct puffs_framebuf **pbp*);

DESCRIPTION

IMPORTANT NOTE! This document describes interfaces which are not yet guaranteed to be stable. In case you update your system sources, please recompile everything and fix compilation errors. If your sources are out-of-sync, incorrect operation may result. The interfaces in this document will most likely be hugely simplified in later versions or made transparent to the implementation.

puffs provides a framework for creating file systems as userspace servers. Operations are transported from the kernel virtual file system layer to the concrete implementation behind **puffs**, where they are processed and results are sent back to the kernel.

It is possible to use **puffs** in two different ways. Calling **puffs_mainloop**() takes execution context away from the caller and automatically handles all requests by using the callbacks. By using puffs_framebuf(3) in conjuction with **puffs_mainloop**(), it is possible to handle I/O to and from file descriptors. This is suited e.g. for distributed file servers.

Library operation

Operations on the library always require a pointer to the opaque context identifier, *struct puffs_usermount*. It is obtained by calling **puffs_init**().

puffs operates using operation callbacks. They can be initialized using the macro **PUFFSOP_SET**(*pops*, *fsname*, *type*, *opname*), which will initialize the operation **puffs_type_opname**() in *pops* to **fsname_type_opname**(). All operations are initialized to a default state with the call **PUFFSOP_INIT**(*pops*). All of the VFS routines are mandatory, but all of the node operations with the exception of **puffs_node_lookup**() are optional. However, leaving operations blank will naturally have an effect on the features available from the file system implementation.

puffs_init(*pops*, *mntfromname*, *puffsname*, *private*, *flags*)

Initializes the library context. *pops* specifies the callback operations vector. *mntfromname* is device the file system is mounted from. This can be for example a block device such as /dev/wd0a or, if the file system is pseudo file system, the **puffs** device name can be given by _PATH_PUFFS. This value is used for example in the first column of the output of mount(8) and df(1). *puffsname* is the file system type. It will always be prepended with the string "puffs|". If possible, file server binaries should be named using the format "mount_myfsnamehere" and this value should equal "myfsnamehere". A file system specific context pointer can optionally be given in *private*. This can be retrieved by **puffs_getspecific**(). Flags for **puffs** can be given via *pflags*. Currently the following flags are supported:

PUFFS_KFLAG_NOCACHE_NAME — Do not enter pathname components into the name cache. This means that every time the kernel does a lookup for a componentname, the file server will be consulted.

PUFFS_KFLAG_NOCACHE_PAGE	Do not use the page cache. This means that all reads and writes to regular file are propagated to the file server for handling. This option makes a difference only for regular files.
PUFFS_KFLAG_NOCACHE	An alias for both PUFFS_KFLAG_NOCACHE_NAME and PUFFS_KFLAG_NOCACHE_PAGE.
PUFFS_KFLAG_ALLOPS	This flag requests that all operations are sent to userspace. Normally the kernel shortcircuits unimplemented operations. This flag is mostly useful for debugging purposes.
PUFFS_KFLAG_WTCACHE	Set the file system cache behavior as write-through. This means that all writes are immediately issued to the file server instead of being flushed in file system sync. This is useful especially for distributed file systems.
PUFFS_KFLAG_IAONDEMAND	Issue inactive only on demand. If a file server defines the inactive method, call it only if the file server has explicitly requested that inactive be called for the node in question. Once inactive has been called for a node, it will not be called again unless the request to call inactive is reissued by the file server. See **puffs_setback**() in puffs_ops(3) for more information.
PUFFS_KFLAG_LOOKUP_FULLPNBUF	This flag affects only the parameter *pcn* to **puffs_node_lookup**(). If this flag is not given, only the next pathname component under lookup is found from *pcn->pcn_name*. If this flag is given, the full path the kernel was asked to resolve can be found from there.
PUFFS_FLAG_BUILDPATH	The framework will build a complete path name, which is supplied with each operation and can be found from the *pn_po.po_full_pcn* field in a *struct puffs_node*. The option assumes that the framework can map a cookie to a *struct puffs_node*. See **Cookies** for more information on cookie mapping. See puffs_path(3) for more information on library calls involving paths.
PUFFS_FLAG_HASHPATH	Calculate a hash of the path into the path object field *po_hash*. This hash value is used by **puffs_path_walkcmp**() to avoid doing a full comparison for every path equal in length to the one searched for. Especially if the file system uses the abovementioned function, it is a good idea to define this flag.
PUFFS_FLAG_OPDUMP	This option makes the framework dump a textual representation of each operation before executing it. It is useful for debugging purposes.

The following functions can be used to query or modify the global state of the file system. Note, that all calls are not available at all times.

puffs_getselectable(*pu*)
Returns a handle to do I/O multiplexing with: select(2), poll(2), and kqueue(2) are all examples of acceptable operations.

puffs_setblockingmode(*pu*, *mode*)
Sets the file system upstream access to blocking or non-blocking mode. Acceptable values for the argument are PUFFSDEV_BLOCK and PUFFSDEV_NONBLOCK.

This routine can be called only after calling **puffs_mount**().

puffs_getstate(*pu*)
Returns the state of the file system. It is maintained by the framework and is mostly useful for the framework itself. Possible values are PUFFS_STATE_BEFOREMOUNT, PUFFS_STATE_RUNNING, PUFFS_STATE_UNMOUNTING and PUFFS_STATE_UNMOUNTED.

puffs_setstacksize(*pu*, *stacksize*)
Sets the stack size used when running callbacks. The default is PUFFS_STACKSIZE_DEFAULT bytes of stack space per request. The minimum stacksize is architecture-dependent and can be specified by using the opaque constant PUFFS_STACKSIZE_MIN.

puffs_setroot(*pu*, *node*)
Sets the root node of mount *pu* to *node*. Setting the root node is currently required only if the path framework is used, see puffs_path(3).

puffs_setrootinfo(*pu*, *vt*, *vsize*, *rdev*)
The default root node is a directory. In case the file system wants something different, it can call this function and set the type, size and possible device type to whatever it wants. This routine is independent of **puffs_setroot**().

puffs_getroot(*pu*)
Returns the root node set earlier.

puffs_getspecific(*pu*)
Returns the *private* argument of **puffs_init**().

puffs_setmaxreqlen(*pu*, *maxreqlen*)
In case the file system desires a maximum buffer length different from the default, the amount *maxreqlen* will be requested from the kernel when the file system is mounted.

It is legal to call this function only between **puffs_init**() and **puffs_mount**().

NOTE This does not currently work.

puffs_getmaxreqlen(*pu*)
Returns the maximum request length the kernel will need for a single request.

NOTE This does not currently work.

puffs_setfhsize(*pu*, *fhsize*, *flags*)
Sets the desired file handle size. This must be called if the file system wishes to support NFS exporting file systems of the **fh***() family of function calls.

In case all nodes in the file system produce the same length file handle, it must be supplied as *fhsize*. In this case, the file system may ignore the length parameters in the file handle callback routines, as the kernel will always pass the correct length buffer. However, if the file handle size varies according to file, the argument *fhsize* defines the maximum size of a file handle for the file system. In this case the file system must take care of the handle lengths by itself in the file handle callbacks, see puffs_ops(3) for more information. Also, the flag PUFFS_FHFLAG_DYNAMIC must be provided in the argument *flags*.

In case the file system wants to sanity check its file handle lengths for the limits of NFS, it can supply PUFFS_FHFLAG_NFSV2 and PUFFS_FHFLAG_NFSV3 in the *flags* parameter. It is especially important to note that these are not directly the limits specified by the protocols, as the kernel uses some bytes from the buffer space. In case the file handles are too large, mount will return an error.

It is legal to call this function only between **puffs_init**() and **puffs_mount**().

puffs_setncookiehash(*pu*, *ncookiehash*)

The parameter *ncookiehash* controls the amount of hash buckets the kernel has for reverse lookups from cookie to vnode. Technically the default is enough, but a memory/time tradeoff can be made by increasing this for file systems which know they will have very many active files.

It is legal to call this function only between **puffs_init**() and **puffs_mount**().

After the correct setup for the library has been established and the backend has been initialized the file system is made operational by calling **puffs_mount**(). After this function returns the file system should start processing requests.

puffs_mount(*pu*, *dir*, *mntflags*, *root_cookie*)

pu is the library context pointer from **puffs_init**(). The argument *dir* signifies the mount point and *mntflags* is the flagset given to mount(2). The value *root_cookie* will be used as the cookie for the file system root node.

Using the built-in eventloop

puffs_ml_loop_fn(*pu*)

Loop function signature.

puffs_ml_setloopfn(*pu*, *lfn*)

Set loop function to *lfn*. This function is called once each time the event loop loops. It is not a well-defined interval, but it can be made fairly regular by setting the loop timeout by **puffs_ml_settimeout**().

puffs_ml_settimeout(*pu*, *ts*)

Sets the loop timeout to *ts* or disables it if *ts* is NULL. This can be used to roughly control how often the loop callback **lfn**() is called

puffs_daemon(*pu*, *nochdir*, *noclose*)

Detach from the console like **daemon**(*3*). This call synchronizes with **puffs_mount**() and the foreground process does not exit before the file system mount call has returned from the kernel.

puffs_mainloop(*pu*, *flags*)

Handle all requests automatically until the file system is unmounted. It returns 0 if the file system was successfully unmounted or −1 if it was killed in action.

In case puffs_framebuf(3) has been initialized, I/O from the relevant descriptors is processed automatically by the eventloop.

puffs_dispatch_create(*pu*, *pb*, *pccp*)

puffs_dispatch_exec(*pcc*, *pbp*)

In case the use of **puffs_mainloop**() is not possible, requests may be dispatched manually. However, as this is less efficient than using the mainloop, it should never be the first preference.

Calling **puffs_dispatch_create**() creates a dispatch request. The argument *pb* should contains a valid request and upon success *pccp* will contain a valid request context. This context is passed to **puffs_dispatch_exec**() to execute the request. If the request yielded before completing, the routine returns 0, otherwise 1. When the routine completes, *pcc* is made invalid and a pointer to the processed buffer is placed in *pbp*. It is the responsibility of the caller to send the

response (if necessary) and destroy the buffer.

See puffs_cc(3) and puffs_framebuf(3) for further information.

Cookies

Every file (regular file, directory, device node, ...) instance is attached to the kernel using a cookie. A cookie should uniquely map to a file during its lifetime. If file instances are kept in memory, a simple strategy is to use the virtual address of the structure describing the file. The cookie can be recycled when **puffs_node_reclaim**() is called for a node.

For some operations (such as building paths) the framework needs to map the cookie to the framework-level structure describing a file, *struct puffs_node*. It is advisable to simply use the *struct puffs_node* address as a cookie and store file system specific data in the private portion of *struct puffs_node*. The library assumes this by default. If it is not desirable, the file system implementation can call **puffs_set_cookiemap**() to provide an alternative cookie-to-node mapping function.

SEE ALSO

mount(2), puffs_cc(3), puffs_cred(3), puffs_flush(3), puffs_framebuf(3), puffs_node(3), puffs_ops(3), puffs_path(3), puffs_suspend(3), refuse(3), puffs(4)

Antti Kantee, "puffs - Pass-to-Userspace Framework File System", *Proceedings of AsiaBSDCon 2007*, pp. 29-42, March 2007.

Antti Kantee, *Using puffs for Implementing Client-Server Distributed File Systems*, *Helsinki University of Technology*, Tech Report TKK-TKO-B157, September 2007.

Antti Kantee and Alistair Crooks, "ReFUSE: Userspace FUSE Reimplementation Using puffs", *EuroBSDCon 2007*, September 2007.

HISTORY

An unsupported experimental version of **puffs** first appeared in NetBSD 4.0.

AUTHORS

Antti Kantee ⟨pooka@iki.fi⟩

BUGS

Under construction.

NAME

puffs_cc — puffs continuation routines

LIBRARY

library “libpuffs”

SYNOPSIS

```
#include <puffs.h>

void
puffs_cc_yield(struct puffs_cc *pcc);

void
puffs_cc_continue(struct puffs_cc *pcc);

void
puffs_cc_schedule(struct puffs_cc *pcc);

struct puffs_cc *
puffs_cc_getcc(struct puffs_usermount *pu);
```

DESCRIPTION

IMPORTANT NOTE! This document describes interfaces which are not yet guaranteed to be stable. In case you update your system sources, please recompile everything and fix compilation errors. If your sources are out-of-sync, incorrect operation may result. The interfaces in this document will most likely be hugely simplified in later versions or made transparent to the implementation.

These routines are used for the cooperative multitasking suite present in puffs.

puffs_cc_yield(*pcc*)
: Suspend and save the current execution context and return control to the previous point. In practice, from the file system author perspective, control returns back to where either the mainloop or where **puffs_dispatch_exec**() was called from.

puffs_cc_continue(*pcc*)
: Will suspend current execution and return control to where it was before before calling **puffs_cc_yield**(). This is rarely called directly but rather through **puffs_dispatch_exec**().

puffs_cc_schedule(*pcc*)
: Schedule a continuation. As opposed to **puffs_cc_continue**() this call returns immediately. *pcc* will be scheduled sometime in the future.

puffs_cc_getcc(*pu*)
: Returns the current pcc or `NULL` if this is the main thread. *NOTE*: The argument *pu* will most likely disappear at some point.

Before calling **puffs_cc_yield**() a file system will typically want to record some cookie value into its own internal bookkeeping. This cookie should be hooked to the *pcc* so that the correct continuation can be continued when the event it was waiting for triggers. Alternatively, the `puffs_framebuf`(3) framework and **puffs_mainloop**() can be used for handling this automatically.

SEE ALSO

`puffs`(3), `puffs_framebuf`(3)

NAME

puffs_cred — puffs credential and access control routines

LIBRARY

library “libpuffs”

SYNOPSIS

```
#include <puffs.h>

int
puffs_cred_getuid(const struct puffs_cred *pcr, uid_t *uid);

int
puffs_cred_getgid(const struct puffs_cred *pcr, gid_t *gid);

int
puffs_cred_getgroups(const struct puffs_cred *pcr,gid_t *gids,
    short *ngids);

bool
puffs_cred_isuid(const struct puffs_cred *pcr, uid_t uid);

bool
puffs_cred_hasgroup(const struct puffs_cred *pcr, gid_t gid);

bool
puffs_cred_iskernel(const struct puffs_cred *pcr);

bool
puffs_cred_isfs(const struct puffs_cred *pcr);

bool
puffs_cred_isjuggernaut(const struct puffs_cred *pcr);

int
puffs_access(enum vtype type,mode_t file_mode,uid_t owner,gid_t group,
    mode_t access_mode,const struct puffs_cred *pcr);

int
puffs_access_chown(uid_t owner,gid_t group,uid_t newowner,gid_t newgroup,
    const struct puffs_cred *pcr);

int
puffs_access_chmod(uid_t owner,gid_t group,enum vtype type,
    mode_t newmode,const struct puffs_cred *pcr);

int
puffs_access_times(uid_t owner,gid_t group,mode_t file_mode,
    int va_utimes_null,const struct puffs_cred *pcr);
```

DESCRIPTION

IMPORTANT NOTE! This document describes interfaces which are not yet guaranteed to be stable. In case you update your system sources, please recompile everything and fix compilation errors. If your sources are out-of-sync, incorrect operation may result. The interfaces in this document will most likely be hugely simplified in later versions or made transparent to the implementation.

These functions can be used to check operation credentials and perform access control. The structure *struct puffs_cred* can contain two types of credentials: ones belonging to users and ones belonging to

the kernel. The latter is further divided into generic kernel credentials and file system credentials. The general rule is that these should be treated as more powerful than superuser credentials, but the file system is free to treat them as it sees fit.

Credentials

The **puffs_cred_get**() family of functions fetch the uid or gid(s) from the given credential cookie. They return 0 for success or −1 for an error and set *errno*. An error happens when the credentials represent kernel or file system credentials and do not contain an uid or gid(s).

For **puffs_cred_getgroups**(), the argument *gids* should point to an array with room for ∗*ngids* elements.

The **puffs_cred_is**() family of functions return 1 if the truth value of the function for *pcr* is true and 0 if it is false. The function **puffs_cred_isjuggernaut**() is true if *pcr* describes superuser, kernel or file system credentials.

Access Control

To help the programmers task of emulating normal kernel file system access control semantics, several helper functions are provided to check if access is allowed. They return 0 if access should be permitted or an errno value to indicate that access should be denied with the returned value.

puffs_access() is equivalent to the kernel **vaccess**() function. The arguments specify current information of the file to be tested with the exception of *access_mode*, which is a combination of PUFFS_VREAD, PUFFS_VWRITE and PUFFS_VEXEC indicating the desired file access mode.

The rest of the functions provide UFS semantics for operations. **puffs_access_chown**() checks if it is permissible to chown a file with the current uid and gid to the new uid and gid with the credentials of *pcr*.

puffs_access_chmod() checks against permission to chmod a file of type *type* to the mode *newmode*.

Finally, **puffs_access_times**() checks if it is allowable to update the timestamps of a file. The argument *va_utimes_null* signals if the flags VA_UTIMES_NULL was set in *va_vaflags* of *struct vattr*. If coming from a path where this information is unavailable, passing 0 as this argument restricts the permission of modification to the owner and superuser. Otherwise the function checks for write permissions to the node and returns success if the caller has write permissions.

SEE ALSO

puffs(3), vnode(9)

NAME

puffs_flush — puffs kernel cache flushing and invalidation routines

LIBRARY

library “libpuffs”

SYNOPSIS

```
#include <puffs.h>

int
puffs_inval_namecache_dir(struct puffs_usermount *pu, void *cookie);

int
puffs_inval_namecache_all(struct puffs_usermount *pu);

int
puffs_inval_pagecache_node(struct puffs_usermount *pu, void *cookie);

int
puffs_inval_pagecache_node_range(struct puffs_usermount *pu,void *cookie,
    off_t start,off_t end);

int
puffs_flush_pagecache_node(struct puffs_usermount *pu, void *cookie);

int
puffs_flush_pagecache_node_range(struct puffs_usermount *pu,void *cookie,
    off_t start,off_t end);
```

DESCRIPTION

IMPORTANT NOTE! This document describes interfaces which are not yet guaranteed to be stable. In case you update your system sources, please recompile everything and fix compilation errors. If your sources are out-of-sync, incorrect operation may result. The interfaces in this document will most likely be hugely simplified in later versions or made transparent to the implementation.

These routines are used inform the kernel that any information it might have cached is no longer valid. **puffs_inval_namecache_dir**() invalidates the name cache for a given directory. The argument *cookie* should describe an existing and valid directory cookie for the file system. Similarly, **puffs_inval_namecache_all**() invalidates the name cache for the entire file system (this routine might go away).

The cached pages (file contents) for a regular file described by *cookie* are invalidated using **puffs_inval_pagecache_node**(). A specific range can be invalidated using **puffs_inval_pagecache_node_range**() for a platform specific page level granularity. The offset *start* will be *truncated* to a page boundary while *end* will be *rounded up* to the next page boundary. As a special case, specifying 0 as *end* will invalidate all contents from *start* to the end of the file.

It is especially important to note that these routines will not only invalidate data in the "read cache", but also data in the "write back" cache (conceptually speaking; in reality they are the same cache), which has not yet been flushed to the file server. Therefore any unflushed data will be lost.

The counterparts of the invalidation routines are the flushing routines **puffs_flush_pagecache_node**() and **puffs_flush_pagecache_node_range**(), which force unwritten data from the kernel page cache to be written. For the flush range version, the same range rules as with the invalidation routine apply. The data is flushed asynchronously, i.e. if the routine returns successfully, all the caller knows is that the data has been queued for writing.

SEE ALSO

`puffs`(3)

NAME

puffs_framebuf — buffering and event handling for networked file systems

LIBRARY

library “libpuffs”

SYNOPSIS

```
#include <puffs.h>

struct puffs_framebuf *
puffs_framebuf_make();

void
puffs_framebuf_destroy(struct puffs_framebuf *pufbuf);

void
puffs_framebuf_recycle(struct puffs_framebuf *pufbuf);

int
puffs_framebuf_reserve_space(struct puffs_framebuf *pufbuf, size_t space);

int
puffs_framebuf_putdata(struct puffs_framebuf *pufbuf,const void *data,
    size_t dlen);

int
puffs_framebuf_putdata_atoff(struct puffs_framebuf *pufbuf,size_t offset,
    const void *data,size_t dlen);

int
puffs_framebuf_getdata(struct puffs_framebuf *pufbuf,void *data,
    size_t dlen);

int
puffs_framebuf_getdata_atoff(struct puffs_framebuf *pufbuf,size_t offset,
    void *data,size_t dlen);

size_t
puffs_framebuf_telloff(struct puffs_framebuf *pufbuf);

size_t
puffs_framebuf_tellsize(struct puffs_framebuf *pufbuf);

size_t
puffs_framebuf_remaining(struct puffs_framebuf *pufbuf);

int
puffs_framebuf_seekset(struct puffs_framebuf *pufbuf, size_t offset);

int
puffs_framebuf_getwindow(struct puffs_framebuf *pufbuf,size_t offset,
    void **winp,size_t *winlen);

int
puffs_framev_enqueue_cc(struct puffs_cc *pcc,int fd,
    struct puffs_framebuf *pufbuf,int flags);

void
puffs_framev_cb(struct puffs_usermount *pu,int fd,
```

*struct puffs_framebuf *pufbuf*, *void *arg*, *int flags*, *int error*);

void
puffs_framev_enqueue_cb(*struct puffs_usermount *pu*, *int fd*,
*struct puffs_framebuf *pufbuf*, *puffs_framebuf_cb fcb*, *void *fcb_arg*,
int flags);

void
puffs_framev_enqueue_justsend(*struct puffs_usermount *pu*, *int fd*, *struct*,
puffs_framebuf, **pufbuf"*, *int waitreply*, *int flags*);

void
puffs_framev_enqueue_directsend(*struct puffs_usermount *pu*, *int fd*,
*struct puffs_framebuf *pufbuf*, *int flags*);

void
puffs_framev_enqueue_directreceive(*struct puffs_usermount *pu*, *int fd*,
*struct puffs_framebuf *pufbuf*, *int flags*);

int
puffs_framev_framebuf_ccpromote(*struct puffs_framebuf *pufbuf*,
*struct puffs_cc *pcc*);

int
puffs_framev_enqueue_waitevent(*struct puffs_cc *pcc*, *int fd*, *int *what*);

int
puffs_framev_readframe_fn(*struct puffs_usermount *pu*,
*struct puffs_framebuf *pufbuf*, *int fd*, *int *done*);

int
puffs_framev_writeframe_fn(*struct puffs_usermount *pu*,
*struct puffs_framebuf *pufbuf*, *int fd*, *int *done*);

int
puffs_framev_cmpframe_fn(*struct puffs_usermount *pu*,
*struct puffs_framebuf *cmp1*, *struct puffs_framebuf *cmp2*,
*int *notresp*);

void
puffs_framev_gotframe_fn(*struct puffs_usermount *pu*,
*struct puffs_framebuf *pufbuf*);

void
puffs_framev_fdnotify_fn(*struct puffs_usermount *pu*, *int fd*, *int what*);

void
puffs_framev_init(*struct puffs_usermount *pu*,
puffs_framev_readframe_fn rfb, *puffs_framev_writeframe_fn wfb*,
puffs_framev_cmpframe_fn cmpfb, *puffs_framev_gotframe_fn gotfb*,
puffs_framev_fdnotify_fn fdnotfn);

int
puffs_framev_addfd(*struct puffs_usermount *pu*, *int fd*, *int what*);

int
puffs_framev_enablefd(*struct puffs_usermount *pu*, *int fd*, *int what*);

int
puffs_framev_disablefd(*struct puffs_usermount *pu*, *int fd*, *int what*);

int
puffs_framev_removefd(*struct puffs_usermount *pu*, *int fd*, *int error*);

void
puffs_framev_unmountonclose(*struct puffs_usermount *pu*, *int fd*, *int what*);

DESCRIPTION

IMPORTANT NOTE! This document describes interfaces which are not yet guaranteed to be stable. In case you update your system sources, please recompile everything and fix compilation errors. If your sources are out-of-sync, incorrect operation may result.

The **puffs_framebuf** routines provide buffering and an event loop structured around the buffers. It operates on top of the puffs continuation framework, puffs_cc(3), and multiplexes execution automatically to an instance whenever one is runnable.

The file system is entered in three different ways:

- An event arrives from the kernel and the puffs_ops(3) callbacks are called to start processing the event.
- A file system which has sent out a request receives a response. Execution is resumed from the place where the file system yielded.
- A request from a peer arrives. A request is an incoming PDU which is not a response to any outstanding request.

puffs_framebuf is used by defining various callbacks and providing I/O descriptors, which are then monitored for activity by the library. A descriptor, when present, can be either enabled or disabled for input and output. If a descriptor is not enabled for a certain direction, the callbacks will not be called even if there were activity on the descriptor. For example, even if a network socket has been added and there is input data in the socket buffer, the read callback will be called only if the socket has been enabled for reading.

File descriptors are treated like sockets: they have two sides, a read side and a write side. The framework determines that one side of the descriptor has been closed if the supplied I/O callbacks return an error or if the I/O multiplexing call says a side has been closed. It is still possible, from the framework perspective, to write to a file descriptor whose read side is closed. However, it is not possible to wait for a response on such a file descriptor. Conversely, it is possible to read responses from a descriptor whose write side is closed. It should be stressed that the implementation underlying the file descriptor might not support this.

The following callbacks can be defined, cf. **puffs_framev_init**(), and all are optional. None of them should block, because this would cause the entire file server to block. One option is to make the descriptors non-blocking before adding them.

rfb — Read a frame from the file descriptor onto the specified buffer.

wfb — Write a frame from the the specified buffer into the file descriptor.

cmpfb — Identify if a buffer is the response to the specified buffer.

gotfb — Called iff no outstanding request matches the incoming frame. In other words, this is called when we receive a request from a peer.

fdnotfn — Receive notifications about a change-of-state in a file descriptor's status.

Better descriptions for each callback are given below.

The buffers of **puffs_framebuf** provide automatic memory management of buffers for the file servers. They provide a cursor to the current buffer offset. Reading or writing data through the normal routines will advance that cursor. Additionally, the buffer size is provided to the user. It represents the maximum offset where data was written.

Generally the write functions will fail if the cannot allocate enough memory to satisfy the buffer length requirements. Read functions will fail if the amount of data written to the buffer is not large enough to satisfy the read.

puffs_framebuf_make()
Create a buffer. Return the address of the buffer or NULL in case no memory was available.

puffs_framebuf_destroy(*pufbuf*)
Free memory used by buffer.

puffs_framebuf_recycle(*pufbuf*)
Reset offsets so that buffer can be reused. Does not free memory or reallocate memory.

puffs_framebuf_reserve_space(*pufbuf*, *space*)
Make sure that the buffer has *space* bytes of available memory starting from the current offset. This is not strictly necessary, but can be used for optimizations where it is known in advance how much memory will be required.

puffs_framebuf_putdata(*pufbuf*, *data*, *dlen*)
Write *dlen* amount of data from the address *data* into the buffer. Moves the offset cursor forward *dlen* bytes.

puffs_framebuf_putdata_atoff(*pufbuf*, *offset*, *data*, *dlen*)
Like **puffs_framebuf_putdata**(), except writes data at buffer offset *offset*. It is legal to write past the current end of the buffer. Does NOT modify the current offset cursor.

puffs_framebuf_getdata(*pufbuf*, *data*, *dlen*)
Read *dlen* bytes of data from the buffer into *data*. Advances the offset cursor.

puffs_framebuf_getdata_atoff(*pufbuf*, *offset*, *data*, *dlen*)
Read data from buffer position *offset*. Does NOT modify the offset cursor.

puffs_framebuf_telloff(*pufbuf*)
Return the offset into the buffer.

puffs_framebuf_tellsize(*pufbuf*)
Return the maximum offset where data has been written, i.e. buffer size.

puffs_framebuf_remaining(*pufbuf*)
Distance from current offset to the end of the buffer, i.e. size-offset.

puffs_framebuf_seekset(*pufbuf*, *offset*)
Set the offset cursor to the position *offset*. This does NOT modify the buffer size, but reserves at least enough memory memory for a write to *offset* and will fail if memory cannot be allocated.

puffs_framebuf_getwindow(*pufbuf*, *offset*, *winp*, *winlen*)
Get a direct memory window into the buffer starting from *offset*. The maximum mapped window size will be *winlen* bytes, but this routine might return a smaller window and the caller should always check the actual mapped size after the call. The window is returned in *winp*. This function not modify the buffer offset, but it DOES set the buffer size to *offset* + *winlen* in case that value is greater than the current size. The window is valid until the next until the next **puffs_framebuf**() call operating on the buffer in question.

puffs_framev_enqueue_cc(*pcc*, *fd*, *pufbuf*, *flags*)
Add the buffer *pufbuf* to outgoing queue of descriptor *fd* and yield with the continuation *pcc*. Execution is resumed once a response is received. Returns 0 if the buffer was successfully enqueued (not necessarily delivered) and non-zero to signal a non-recoverable error.

Usually the buffer is placed at the end of the output queue. However, if *flags* contains PUFFS_FBQUEUE_URGENT, *pufbuf* is placed in the front of the queue to be sent immediately after the current PDU (if any) has been sent.

puffs_framev_enqueue_cb(*pu*, *fd*, *pufbuf*, *fcb*, *fcb_arg*, *flags*)
Enqueue the buffer *pufbuf* for outgoing data and immediately return. Once a response arrives, the callback **fcb**() will be called with the argument *fcb_arg*. The callback function **fcb**() is responsible for freeing the buffer. Returns 0 if the buffer was successfully enqueued (not necessarily delivered) and non-zero to signal a non-recoverable error.

See **puffs_framev_enqueue_cc**() for *flags*.

puffs_framev_cb(*pu*, *pufbuf*, *arg*, *error*)
Callback function. Called when a response to a specific request arrives from the server. If *error* is non-zero, the framework was unable to obtain a response and the function should not examine the contents of *pufbuf*, only do resource cleanup. May not block.

puffs_framev_enqueue_justsend(*pu*, *fd*, *pufbuf*, *waitreply*, *flags*)
Enqueue the buffer *pufbuf* for outgoing traffic and immediately return. The parameter *waitreply* can be used to control if the buffer is to be freed immediately after sending of if a response is expected and the buffer should be freed only after the response arrives (receiving an unexpected message from the server is treated as an error). Returns 0 if the buffer was successfully enqueued (not necessarily delivered) and non-zero to signal a non-recoverable error.

See **puffs_framev_enqueue_cc**() for *flags*.

puffs_framev_enqueue_directsend(*pcc*, *fd*, *pufbuf*, *flags*)
Acts like **puffs_framev_enqueue_justsend**() with the exception that the call yields until the frame has been sent. As opposed to **puffs_framev_enqueue_cc**(), the routine does not wait for input, but returns immediately after sending the frame.

See **puffs_framev_enqueue_cc**() for *flags*.

puffs_framev_enqueue_directreceive(*pcc*, *fd*, *pufbuf*, *flags*)
Receive data into *pufbuf*. This routine yields until a complete frame has been read into the buffer by the readframe routine.

See **puffs_framev_enqueue_cc**() for *flags*.

puffs_framev_framebuf_ccpromote(*pufbuf*, *pcc*)
Promote the framebuffer *pufbuf* sent with **puffs_framev_enqueue_cb**() or **puffs_framev_enqueue_justsend**() to a wait using *pcc* and yield until the result arrives. The response from the file server for *pufbuf* must not yet have arrived. If sent with **puffs_framev_enqueue_justsend**(), the call must be expecting a response.

puffs_framev_enqueue_waitevent(*pcc*, *fd*, *what*)
Waits for an event in *what* to happen on file descriptor *fd*. The events which happened are returned back in *what*. The possible events are PUFFS_FBIO_READ, PUFFS_FBIO_WRITE, and PUFFS_FBIO_ERROR, specifying read, write and error conditions, respectively. Error is always checked.

This call does not depend on if the events were previously enabled on the file descriptor - the specified events are always checked regardless.

There is currently no other way to cancel or timeout a call except by removing the file descriptor in question. This may change in the future.

puffs_framev_readframe_fn(*pu*, *pufbuf*, *fd*, *done*)
Callback function. Read at most one frame from file descriptor *fd* into the buffer *pufbuf*. If a complete frame is read, the value pointed to by *done* must be set to 1. This function should return 0 on success (even if a complete frame was not yet read) and a non-zero `errno` to signal a fatal error. In case a fatal error is returned, the read side of the file descriptor is marked closed. This routine will be called with the same buffer argument until a complete frame has been read. May not block.

puffs_framev_writeframe_fn(*pu*, *pufbuf*, *fd*, *done*)
Write the frame contained in *pufbuf* to the file descriptor *fd*. In case the entire frame is successfully written, ∗*done* should be set to 1. This function should return 0 on success (even if a complete frame was not yet written) and a non-zero `errno` to signal a fatal error. In case a fatal error is returned, the write side of the file descriptor is marked closed. This routine will be called with the same buffer argument until the complete frame has been written. May not block.

It is a good idea to make sure that this function can handle a possible `SIGPIPE` caused by a closed connection. For example, the file server can opt to trap `SIGPIPE` or, if writing to a socket, call **send**() with the flag `MSG_NOSIGNAL` instead of using **write**().

puffs_framev_cmpframe_fn(*pu*, *pufbuf_cmp1*, *pufbuf_cmp2*, *notresp*)
Compare the file system internal request tags in *pufbuf_cmp1* and *pufbuf_cmp2*. Should return 0 if the tags are equal, 1 if first buffer's tag is greater than the second and −1 if it is smaller. The definitions "greater" and "smaller" are used transparently by the library, e.g. like `qsort`(3). If it can be determined from *pufbuf_cmp1* that it is not a response to any outstanding request, *notresp* should be set to non-zero. This will cause **puffs_framebuf** to skip the test of the buffer against the rest of the outstanding request. May not block.

puffs_framev_gotframe_fn(*pu*, *pufbuf*)
Called when no outstanding request matches an incoming frame. The ownership of *pufbuf* is transferred to the called function and must be destroyed once processing is over. May not block.

puffs_framev_fdnotify_fn(*pu*, *fd*, *what*)
Is called when the read or write side of the file descriptor *fd* is closed. It is called once for each side, the bitmask parameter *what* specified what is currently closed: `PUFFS_FBIO_READ` and `PUFFS_FBIO_WRITE` for read and write, respectively.

puffs_framev_init(*pu*, *rfb*, *wfb*, *cmpfb*, *gotfb*, *fdnotfn*)
Initializes the given callbacks to the system. They will be used when **puffs_mainloop**() is called. The framework provides the routines **puffs_framev_removeonclose**() and **puffs_framev_unmountonclose**(), which can be given as *fdnotfn*. The first one removes the file descriptor once both sides are closed while the second one unmounts the file system and exits the mainloop.

puffs_framev_addfd(*pu*, *fd*, *what*)
Add file descriptor *fd* to be handled by the framework. It is legal to add a file descriptor either before calling **puffs_mainloop**() or at time when running. The parameter *what* controls enabling of input and output events and can be a bitwise combination of `PUFFS_FBIO_READ` and `PUFFS_FBIO_WRITE`. If not specified, the descriptor will be in a disabled state.

puffs_framev_enablefd(*pu*, *fd*, *error*)
Enable events of type *what* for file descriptor *fd*.

puffs_framev_disablefd(*pu*, *fd*, *error*)
Disable events of type *what* for file descriptor *fd*.

puffs_framev_removefd(*pu*, *fd*, *error*)
Remove file descriptor *fd* from the list of descriptors handled by the framework. Removing a file descriptor causes all operations blocked either on output or input to be released with the error value *error*. In case 0 is supplied as this parameter, ECONNRESET is used.

The file system *must* explicitly remove each fd it has added. A good place to do this is **puffs_framev_fdnotify_fn**() or **puffs_node_reclaim**(), depending a little on the structure of the file system.

puffs_framev_unmountonclose(*pu*, *fd*, *what*)
This is library provided convenience routine for **puffs_framev_fdnotify_fn**(). It unmounts the file system when both the read and write side are closed. It is useful for file systems such as mount_psshfs(8) which depend on a single connection.

CODE REFERENCES

The current users of **puffs_framebuf** in the tree are mount_psshfs(8) and mount_9p(8). See src/usr.sbin/puffs/mount_psshfs and src/usr.sbin/puffs/mount_9p for the respective usage examples.

RETURN VALUES

These functions generally return −1 to signal error and set errno to indicate the type of error.

SEE ALSO

puffs(3), puffs_cc(3), puffs_ops(3)

Antti Kantee, *Using puffs for Implementing Client-Server Distributed File Systems, Helsinki University of Technology*, Tech Report TKK-TKO-B157, September 2007.

NAME

puffs_node — puffs node routines

LIBRARY

library “libpuffs”

SYNOPSIS

```
#include <puffs.h>

struct puffs_node *
puffs_pn_new(struct puffs_usermount *pu, void *priv);

void *
puffs_nodewalk_fn(struct puffs_usermount *pu,struct puffs_node *pn,
    void *arg);

void *
puffs_pn_nodewalk(struct puffs_usermount *pu,puffs_nodewalk_fn nwfn,
    void *arg);

void
puffs_pn_remove(struct puffs_node *pn);

void
puffs_pn_put(struct puffs_node *pn);
```

DESCRIPTION

IMPORTANT NOTE! This document describes interfaces which are not yet guaranteed to be stable. In case you update your system sources, please recompile everything and fix compilation errors. If your sources are out-of-sync, incorrect operation may result. The interfaces in this document will most likely be hugely simplified in later versions or made transparent to the implementation.

MORE IMPORTANT STUFF! It should especially be noted, that it is yet completely unsure, how much of the internals contents of *struct puffs_node* will be exposed to file systems in the end.

puffs_pn_new(*pu*, *priv*)
: Create a new node and attach it to the mountpoint *pu*. The argument *priv* can be used for associating file system specific data with the new node and will not be accessed by puffs.

puffs_nodewalk_fn(*pu*, *pn*, *arg*)
: A callback for **puffs_nodewalk**(). The list of nodes is iterated in the argument *pn* and the argument *arg* is the argument given to **puffs_nodewalk**().

puffs_nodewalk(*pu*, *nwfn*, *arg*)
: Walk all nodes associted with the mountpoint *pu* and call **nwfn**() for them. The walk is aborted if **puffs_nodewalk_fn**() returns a value which is not NULL. This value is also returned this function. In case the whole set of nodes is traversed, NULL is returned. This function is useful for example in handling the **puffs_fs_sync**() callback, when cached data for every node should be flushed to stable storage.

puffs_pn_remove(*pn*)
: Signal that a node has been removed from the file system, but do not yet release resources associated with the node. This will prevent the nodewalk functions from accessing the node. If necessary, this is usually called from **puffs_node_remove**() and **puffs_node_rmdir**().

puffs_pn_put(*pn*)
Free all resources associated with node *pn*. This is typically called from **puffs_node_reclaim**().

SEE ALSO
puffs(3)

NAME

puffs_ops — puffs callback operations

LIBRARY

library “libpuffs”

SYNOPSIS

```
#include <puffs.h>

int
puffs_fs_statvfs(struct puffs_usermount *pu, struct statvfs *sbp);

int
puffs_fs_sync(struct puffs_usermount *pu, int waitfor,
    const struct puffs_cred *pcr);

int
puffs_fs_fhtonode(struct puffs_usermount *pu, void *fid, size_t fidsize,
    struct puffs_newinfo *pni);

int
puffs_fs_nodetofh(struct puffs_usermount *pu, void *cookie, void *fid,
    size_t *fidsize);

void
puffs_fs_suspend(struct puffs_usermount *pu, int status);

int
puffs_fs_unmount(struct puffs_usermount *pu, int flags);

int
puffs_node_lookup(struct puffs_usermount *pu, void *opc,
    struct puffs_newinfo *pni, const struct puffs_cn *pcn);

int
puffs_node_create(struct puffs_usermount *pu, void *opc,
    struct puffs_newinfo *pni, const struct puffs_cn *pcn,
    const struct vattr *vap);

int
puffs_node_mknod(struct puffs_usermount *pu, void *opc,
    struct puffs_newinfo *pni, const struct puffs_cn *pcn,
    const struct vattr *vap);

int
puffs_node_open(struct puffs_usermount *pu, void *opc, int mode,
    const struct puffs_cred *pcr);

int
puffs_node_close(struct puffs_usermount *pu, void *opc, int flags,
    const struct puffs_cred *pcr);

int
puffs_node_access(struct puffs_usermount *pu, void *opc, int mode,
    const struct puffs_cred *pcr);

int
puffs_node_getattr(struct puffs_usermount *pu, void *opc, struct vattr *vap,
```

```
    const struct puffs_cred *pcr);

int
puffs_node_setattr(struct puffs_usermount *pu, void *opc,
    const struct vattr *vap, const struct puffs_cred *pcr);

int
puffs_node_poll(struct puffs_usermount *pu, void *opc, int *events);

int
puffs_node_mmap(struct puffs_usermount *pu, void *opc, int flags,
    const struct puffs_cred *pcr);

int
puffs_node_fsync(struct puffs_usermount *pu, void *opc,
    const struct puffs_cred *pcr, int flags, off_t offlo, off_t offhi);

int
puffs_node_seek(struct puffs_usermount *pu, void *opc, off_t oldoff,
    off_t newoff, const struct puffs_cred *pcr);

int
puffs_node_remove(struct puffs_usermount *pu, void *opc, void *targ,
    const struct puffs_cn *pcn);

int
puffs_node_link(struct puffs_usermount *pu, void *opc, void *targ,
    const struct puffs_cn *pcn);

int
puffs_node_rename(struct puffs_usermount *pu, void *opc, void *src,
    const struct puffs_cn *pcn_src, void *targ_dir, void *targ,
    const struct puffs_cn *pcn_targ);

int
puffs_node_mkdir(struct puffs_usermount *pu, void *opc,
    struct puffs_newinfo *pni, const struct puffs_cn *pcn,
    const struct vattr *vap);

int
puffs_node_rmdir(struct puffs_usermount *pu, void *opc, void *targ,
    const struct puffs_cn *pcn);

int
puffs_node_readdir(struct puffs_usermount *pu, void *opc,
    struct dirent *dent, off_t *readoff, size_t *reslen,
    const struct puffs_cred *pcr, int *eofflag, off_t *cookies,
    size_t *ncookies);

int
puffs_node_symlink(struct puffs_usermount *pu, void *opc,
    struct puffs_newinfo *pni, const struct puffs_cn *pcn_src,
    const struct vattr *vap, const char *link_target);

int
puffs_node_readlink(struct puffs_usermount *pu, void *opc,
    const struct puffs_cred *pcr, char *link, size_t *linklen);
```

int
puffs_node_read(*struct puffs_usermount *pu*, *void *opc*, *uint8_t *buf*, *off_t offset*, *size_t *resid*, *const struct puffs_cred *pcr*, *int ioflag*);

int
puffs_node_write(*struct puffs_usermount *pu*, *void *opc*, *uint8_t *buf*, *off_t offset*, *size_t *resid*, *const struct puffs_cred *pcr*, *int ioflag*);

int
puffs_node_print(*struct puffs_usermount *pu*, *void *opc*);

int
puffs_node_reclaim(*struct puffs_usermount *pu*, *void *opc*);

int
puffs_node_inactive(*struct puffs_usermount *pu*, *void *opc*);

void
puffs_setback(*struct puffs_cc *pcc*, *int op*);

void
puffs_newinfo_setcookie(*struct puffs_newinfo *pni*, *void *cookie*);

void
puffs_newinfo_setvtype(*struct puffs_newinfo *pni*, *enum vtype vtype*);

void
puffs_newinfo_setsize(*struct puffs_newinfo *pni*, *voff_t size*);

void
puffs_newinfo_setrdev(*struct puffs_newinfo *pni*, *dev_t rdev*);

DESCRIPTION

IMPORTANT NOTE! This document describes interfaces which are not yet guaranteed to be stable. In case you update your system sources, please recompile everything and fix compilation errors. If your sources are out-of-sync, incorrect operation may result. The interfaces in this document will most likely be hugely simplified in later versions or made transparent to the implementation.

The operations **puffs** requires to function can be divided into two categories: file system callbacks and node callbacks. The former affect the entire file system while the latter are targeted at a file or a directory and a file. They are roughly equivalent to the vfs and vnode operations in the kernel.

All callbacks can be prototyped with the file system name and operation name using the macro **PUFFSOP_PROTOS**(*fsname*).

File system callbacks (puffs_fs)

puffs_fs_statvfs(*pu*, *sbp*)

The following fields of the argument *sbp* need to be filled:

```
* unsigned long  f_bsize;     file system block size
* unsigned long  f_frsize;    fundamental file system block size
* fsblkcnt_t     f_blocks;    number of blocks in file system,
*                                 (in units of f_frsize)
*
* fsblkcnt_t     f_bfree;     free blocks avail in file system
* fsblkcnt_t     f_bavail;    free blocks avail to non-root
* fsblkcnt_t     f_bresvd;    blocks reserved for root
```

```
 * fsfilcnt_t     f_files;     total file nodes in file system
 * fsfilcnt_t     f_ffree;     free file nodes in file system
 * fsfilcnt_t     f_favail;    free file nodes avail to non-root
 * fsfilcnt_t     f_fresvd;    file nodes reserved for root
```

puffs_fs_sync(*pu*, *waitfor*, *pcr*)
All the dirty buffers that have been cached at the file server level including metadata should be committed to stable storage. The *waitfor* parameter affects the operation. Possible values are:

MNT_WAIT Wait for all I/O for complete until returning.

MNT_NOWAIT Initiate I/O, but do not wait for completion.

MNT_LAZY Synchorize data not synchoronized by the file system syncer, i.e. data not written when **node_fsync**() is called with FSYNC_LAZY.

The credentials for the initiator of the sync operation are present in *pcr* and will usually be either file system or kernel credentials, but might also be user credentials. However, most of the time it is advisable to sync regardless of the credentials of the caller.

puffs_fs_fhtonode(*pu*, *fid*, *fidsize*, *pni*)
Translates a file handle *fid* to a node. The parameter *fidsize* indicates how large the file handle is. In case the file system's handles are static length, this parameter can be ignored as the kernel guarantees all file handles passed to the file server are of correct length. For dynamic length handles the field should be examined and EINVAL returned in case the file handle length is not correct.

This function provides essentially the same information to the kernel as **puffs_node_lookup**(). The information is necessary for creating a new vnode corresponding to the file handle.

puffs_fs_nodetofh(*pu*, *cookie*, *fid*, *fidsize*)
Create a file handle from the node described by *cookie*. The file handle should contain enough information to reliably identify the node even after reboots and the pathname/inode being replaced by another file. If this is not possible, it is up to the author of the file system to act responsibly and decide if the file system can support file handles at all.

For file systems which want dynamic length file handles, this function must check if the file handle space indicated by *fidsize* is large enough to accommodate the file handle for the node. If not, it must fill in the correct size and return E2BIG. In either case, the handle length should be supplied to the kernel in *fidsize*. File systems with static length handles can ignore the size parameter as the kernel always supplies the correct size buffer.

puffs_fs_suspend(*pu*, *status*)
Called when file system suspension reaches various phases. See puffs_suspend(3) for more information.

puffs_fs_unmount(*pu*, *flags*)
Unmount the file system. The kernel has assumedly flushed all cached data when this callback is executed. If the file system cannot currently be safely be unmounted, for whatever reason, the kernel will honor an error value and not forcibly unmount. However, if the flag MNT_FORCE is not honored by the file server, the kernel will forcibly unmount the file system.

Node callbacks
These operations operate in the level of individual files. The file cookie is always provided as the second argument *opc*. If the operation is for a file, it will be the cookie of the file. The case the operation involves a directory (such as "create file in directory"), the cookie will be for the directory. Some operations take additional cookies to describe the rest of the operands. The return value 0 signals success, else an appropri-

ate errno value should be returned. Please note that neither this list nor the descriptions are complete.

puffs_node_lookup(*pu*, *opc*, *pni*, *pcn*)

This function is used to locate nodes, or in other words translate pathname components to file system data structures. The implementation should match the name in *pcn* against the existing entries in the directory provided by the cookie *opc*. If found, the cookie for the located node should be set in *pni* using **puffs_newinfo_setcookie**(). Additionally, the vnode type and size (latter applicable to regular files only) should be set using **puffs_newinfo_setvtype**() and **puffs_newinfo_setsize**(), respectively. If the located entry is a block device or character device file, the dev_t for the entry should be set using **puffs_newinfo_setrdev**().

The type of operation is found from *pcn->pcn_nameiop*:

PUFFSLOOKUP_LOOKUP	Normal lookup operation.
PUFFSLOOKUP_CREATE	Lookup to create a node.
PUFFSLOOKUP_DELETE	Lookup for node deletion.
PUFFSLOOKUP_RENAME	Lookup for the target of a rename operation (source will be looked up using PUFFSLOOKUP_DELETE).

The final component from a pathname lookup usually requires special treatment. It can be identified by looking at the *pcn->pcn_flags* fields for the flag PUFFSLOOKUP_ISLASTCN. For example, in most cases the lookup operation will want to check if a delete, rename or create operation has enough credentials to perform the operation.

The return value 0 signals a found node and a nonzero value signals an errno. As a special case, ENOENT signals "success" for cases where the lookup operation is PUFFSLOOKUP_CREATE or PUFFSLOOKUP_RENAME. Failure in these cases can be signalled by returning another appropriate error code, for example EACCESS.

Usually a null-terminated string for the next pathname component is provided in *pcn->pcn_name*. In case the file system is using the option PUFFS_KFLAG_LOOKUP_FULLPNBUF, the remainder of the complete pathname under lookup is found in the same location. *pcn->pcn_namelen* always specifies the length of the next component. If operating with a full path, the file system is allowed to consume more than the next component's length in node lookup. This is done by setting *pcn->pcn_consume* to indicate the amount of *extra* characters in addition to *pcn->pcn_namelen* processed.

puffs_node_create(*pu*, *opc*, *pni*, *pcn*, *va*)

puffs_node_mkdir(*pu*, *opc*, *pni*, *pcn*, *va*)

puffs_node_mknod(*pu*, *opc*, *pni*, *pcn*, *va*)

A file node is created in the directory denoted by the cookie *opc* by any of the above callbacks. The name of the new file can be found from *pcn* and the attributes are specified by *va* and the cookie for the newly created node should be set in *pni*. The only difference between these three is that they create a regular file, directory and device special file, respectively.

In case of mknod, the device identifier can be found in *va->va_rdev*.

puffs_node_open(*pu*, *opc*, *mode*, *pcr*)

Open the node denoted by the cookie *opc*. The parameter *mode* specifies the flags that open(2) was called with, e.g. O_APPEND and O_NONBLOCK.

puffs_node_close(*pu*, *opc*, *flags*, *pcr*)

Close a node. The parameter *flags* parameter describes the flags that the file was opened with.

puffs_node_access(*pu*, *opc*, *mode*, *pcr*)
Check if the credentials of *pcr* have the right to perform the operation specified by *mode* onto the node *opc*. The argument *mode* can specify read, write or execute by PUFFS_VREAD, PUFFS_VWRITE, and PUFFS_VEXEC, respectively.

puffs_node_getattr(*pu*, *opc*, *va*, *pcr*)
The attributes of the node specified by *opc* must be copied to the space pointed by *va*.

puffs_node_setattr(*pu*, *opc*, *va*, *pcr*)
The attributes for the node specified by *opc* must be set to those contained in *va*. Only fields of *va* which contain a value different from PUFFS_VNOVAL (typecast to the field's type!) contain a valid value.

puffs_node_poll(*pu*, *opc*, *events*)
Poll for events on node *opc*. If poll(2) events specified in *events* are available, the function should set the bitmask to match available events and return immediately. Otherwise, the function should block (yield) until some events in *events* become available and only then set the *events* bitmask and return.

In case this function returns an error, POLLERR (or it's select(2) equivalent) will be delivered to the calling process.

NOTE! The system call interface for **poll**() contains a timeout parameter. At this level, however, the timeout is not supplied. In case input does not arrive, the file system should periodically unblock and return 0 new events to avoid hanging forever. This will hopefully be better supported by libpuffs in the future.

puffs_node_mmap(*pu*, *opc*, *flags*, *pcr*)
Called when a regular file is being memory mapped by mmap(2). *flags* is currently always 0.

puffs_node_fsync(*pu*, *opc*, *pcr*, *flags*, *offlo*, *offhi*)
Sychronize a node's contents onto stable storage. This is necessary only if the file server caches some information before committing it. The parameter *flags* specifies the minimum level of sychronization required (XXX: they are not yet available). The parameters *offlo* and *offhi* specify the data offsets requiring to be synced. A high offset of 0 means sync from *offlo* to the end of the file.

puffs_node_seek(*pu*, *opc*, *oldoff*, *newoff*, *pcr*)
Test if the node *opc* is seekable to the location *newoff*. The argument *oldoff* specifies the offset we are starting the seek from. Most file systems dealing only with regular will choose to not implement this. However, it is useful for example in cases where files are unseekable streams.

puffs_node_remove(*pu*, *opc*, *targ*, *pcn*)

puffs_node_rmdir(*pu*, *opc*, *targ*, *pcn*)
Remove the node *targ* from the directory indicated by *opc*. The directory entry name to be removed is provided by *pcn*. The rmdir operation removes only directories, while the remove operation removes all other types except directories.

It is paramount to note that the file system may not remove the node data structures at this point, only the directory entry and prevent lookups from finding the node again. This is to retain the UNIX open file semantics. The data may be removed only when **puffs_node_reclaim**() is called for the node, as this assures there are no further users.

puffs_node_link(*pu*, *opc*, *targ*, *pcn*)
Create a hard link for the node *targ* into the directory *opc*. The argument *pcn* provides the directory entry name for the new link.

puffs_node_rename(*pu*, *src_dir*, *src*, *pcn_src*, *targ_dir*, *targ*, *pcn_targ*)
Rename the node *src* with the name specified by *pcn_src* from the directory *src_dir*. The target directory and target name are given by *targ_dir* and *pcn_targ*, respectively. the target node already exists, it is specified by *targ* and must be replaced atomically. Otherwise *targ* is gives as NULL.

It is legal to replace a directory node by another directory node with the means of rename if the target directory is empty, otherwise ENOTEMPTY should be returned. All other types can replace all other types. In case a rename between incompatible types is attempted, the errors ENOTDIR or EISDIR should be returned, depending on the target type.

puffs_node_readdir(*pu*, *opc*, *dent*, *readoff*, *reslen*, *pcr*, *eofflag*, *cookies*, *ncookies*)
To read directory entries, **puffs_node_readdir**() is called. It should store directories as *struct dirent* in the space pointed to by *dent*. The amount of space available is given by *reslen* and before returning it should be set to the amount of space *remaining* in the buffer. The argument *offset* is used to specify the offset to the directory. Its intepretation is up to the file system and it should be set to signal the continuation point when there is no more room for the next entry in *dent*. It is most performant to return the maximal amount of directory entries each call. It is easiest to generate directory entries by using **puffs_nextdent**(), which also automatically advances the necessary pointers.

In case end-of-directory is reached, *eofflag* should be set to one. Note that even a new call to readdir may start where *readoff* points to end-of-directory.

If the file system supports file handles, the arguments *cookies* and *ncookies* must be filled out. *cookies* is a vector for offsets corresponding to read offsets. One cookie should be filled out for each directory entry. The value of the cookie should equal the offset of the *next* directory entry, i.e. which offset should be passed to readdir for the first entry read to be the entry following the current one. *ncookies* is the number of slots for cookies in the cookie vector upon entry to the function and must be set to the amount of cookies stored in the vector (i.e. amount of directory entries read) upon exit. There is always enough space in the cookie vector for the maximal number of entries that will fit into the directory entry buffer. For filling out the vector, the helper function **PUFFS_STORE_DCOOKIE**(*cookies*, *ncookies*, *offset*) can be used. This properly checks against *cookies* being NULL. Note that *ncookies* must be initialized to zero before the first call to **PUFFS_STORE_DCOOKIE**().

puffs_node_symlink(*pu*, *opc*, *pni*, *pcn_src*, *va*, *link_target*)
Create a symbolic link into the directory *opc* with the name in *pcn_src* and the initial attributes in *va*. The argument *link_target* contains a null-terminated string for the link target. The created node cookie should be set in *pni*.

puffs_node_readlink(*pu*, *opc*, *pcr*, *link*, *linklen*)
Read the target of a symbolic link *opc*. The result is placed in the buffer pointed to by *link*. This buffer's length is given in *linklen* and it must be updated to reflect the real link length. A terminating nul character should not be put into the buffer and *must not* be included in the link length.

puffs_node_read(*pu*, *opc*, *buf*, *offset*, *resid*, *pcr*, *ioflag*)
Read the contents of a file *opc*. It will gather the data from *offset* in the file and read the number *resid* octets. The buffer is guaranteed to have this much space. The amount of data requested by *resid* should be read, except in the case of eof-of-file or an error. The parameter *resid* should be set to indicate the amount of request NOT completed. In the normal case this should be 0.

puffs_node_write(*pu*, *opc*, *buf*, *offset*, *resid*, *pcr*, *ioflag*)
puffs_node_write() Write data to a file *opc* at *offset* and extend the file if necessary. The number of octets written is indicated by *resid*; everything must be written or an error will be gener-

ated. The parameter must be set to indicate the amount of data NOT written. In case the flag PUFFS_IO_APPEND is specified, the data should be appended to the end of the file.

puffs_node_print(*pu*, *opc*)
Print information about node. This is used only for kernel-initiated diagnostic purposes.

puffs_node_reclaim(*pu*, *opc*)
The kernel will no longer reference the cookie and resources associated with it may be freed. In case the file *opc* has a link count of zero, it may be safely removed now.

puffs_node_inactive(*pu*, *opc*)
The node *opc* has lost its last reference in the kernel. However, the cookie must still remain valid until **puffs_node_reclaim**() is called.

puffs_setback(*pcc*, *op*)
Issue a "setback" operation which will be handled when the request response is returned to the kernel. Currently this can be only called from mmap, open, remove and rmdir. The valid parameters for *op* are PUFFS_SETBACK_INACT_N1 and PUFFS_SETBACK_INACT_N2. These signal that a file system mounted with PUFFS_KFLAG_IAONDEMAND should call the file system inactive method for the specified node. The node number 1 always means the operation cookie *opc*, while the node number 2 can be used to specify the second node argument present in some methods, e.g. remove.

puffs_newinfo_setcookie(*pni*, *cookie*)
Set cookie for node provided by this method to *cookie*.

puffs_newinfo_setvtype(*pni*, *vtype*)
Set the type of the newly located node to *vtype*. This call is valid only for **lookup**() and **fhtonode**().

puffs_newinfo_setsize(*pni*, *size*)
Set the size of the newly located node to *size*. If left unset, the value defaults to 0. This call is valid only for **lookup**() and **fhtovp**().

puffs_newinfo_setrdev(*pni*, *rdev*)
Set the type of the newly located node to *vtype*. This call is valid only for **lookup**() and **fhtovp**() producing device type nodes.

SEE ALSO

puffs(3), vfsops(9), vnodeops(9)

NAME

puffs_path — puffs pathbuilding routines

LIBRARY

library “libpuffs”

SYNOPSIS

```
#include <puffs.h>

int
pu_pathbuild_fn(struct puffs_usermount *pu,
    const struct puffs_pathobj *po_dir,
    const struct puffs_pathobj *po_comp, size_t offset,
    struct puffs_pathobj *po_new);

int
pu_pathtransform_fn(struct puffs_usermount *pu,
    const struct puffs_pathobj *po_base, const struct puffs_cn *pcn,
    struct puffs_pathobj *po_new);

int
pu_pathcmp_fn(struct puffs_usermount *pu, struct puffs_pathobj *po1,
    struct puffs_pathobj *po2, size_t checklen, int checkprefix);

void
pu_pathfree_fn(struct puffs_usermount *pu, struct puffs_pathobj *po);

int
pu_namemod_fn(struct puffs_usermount *pu, struct puffs_pathobj *po_dir,
    struct puffs_cn *pcn);

struct puffs_pathobj *
puffs_getrootpathobj(struct puffs_usermount *pu);
```

DESCRIPTION

IMPORTANT NOTE! This document describes interfaces which are not yet guaranteed to be stable. In case you update your system sources, please recompile everything and fix compilation errors. If your sources are out-of-sync, incorrect operation may result. The interfaces in this document will most likely be hugely simplified in later versions or made transparent to the implementation.

The puffs library has the ability to provide full pathnames for backends which require them. Normal file systems should be constructed without the file system node tied to a file name and should not used routines described herein. An example of a file system where the backend requires filenames is `mount_psshfs`(8).

The features described here are enabled by passing `PUFFS_FLAG_BUILDPATH` to **puffs_init**(). This facility requires to use puffs nodes to store the contents of the pathname. Either the address of the operation cookie must directly be that of the puffs node, or **puffs_set_cmap**() must be used to set a mapping function from the cookie to the puffs node associated with the cookie. Finally, the root node for the file system must be set using **puffs_setroot**() and the root path object retrieved using **puffs_getrootpathobj**() and initialized.

There are two different places a filename can be retrieved from. It is available for each puffs node after the node has been registered with the framework, i.e. *after* the routine creating the node returns. In other words, there is a window between the node is created and when the pathname is available and multithreaded file systems must take this into account. The second place where a pathname is available is from the componentname *struct puffs_pcn* in operations which are passed one. These can be retrieved using the con-

venience macros **PNPATH**() and **PCNPATH**() for node and componentname, respectively. The type of object they return is *void* *.

By default the framework manages "regular" filenames, which consist of directory names separated by "/" and a final component. If the file system wishes to use pathnames of this format, all it has to do it enable the feature. Everything else, including bookkeeping for node and directory renames, is done by the library. The callback routines described next provide the ability to build non-standard pathnames. A **pu_foo_fn**() callback is set using the **puffs_set_foo**() routine.

This manual page is still unfinished. Please take a number and wait in line.

SEE ALSO

puffs(3), puffs_node(3), mount_psshfs(8), mount_sysctlfs(8)

NAME

puffs_suspend — puffs file system suspension and snapshotting

LIBRARY

library “libpuffs”

SYNOPSIS

```
#include <puffs.h>

int
puffs_fs_suspend(struct puffs_usermount *pu);
```

DESCRIPTION

IMPORTANT NOTE! This document describes interfaces which are not yet guaranteed to be stable. In case you update your system sources, please recompile everything and fix compilation errors. If your sources are out-of-sync, incorrect operation may result. The interfaces in this document will most likely be hugely simplified in later versions or made transparent to the implementation.

The function **puffs_fs_suspend**() requests the kernel to suspend operations to the file system indicated by *pu*. There are several possible outcomes: nothing, an error or success. These will be indicated through the callback of the same name. The file system must set this callback if it wants to be notified of file system suspension. The interface call itself returns 0 on success or −1 on error and sets *errno*. In case an error is returned, the callback will never be called. However, the converse does not hold and the callback may never be called even if the library call is successful.

In case the kernel is successful to start suspending the file system, the callback is called with status `PUFFS_SUSPEND START`. The file system implementation may use this as a hint on how to handle following operations. Once the file system has successfully been suspended, the status `PUFFS_SUSPEND_SUSPENDED` will be delivered. In case there was an error while suspending, `PUFFS_SUSPEND_ERROR` is given. This effectively nullifies any `PUFFS_SUSPEND_START` given earlier. Operation will automatically resume after suspension and the status `PUFFS_SUSPEND_RESUME` is delivered to the callback. Error or success is always provided in case start is given.

The file system is supposed to do a file system specific snapshotting routine when it receives `PUFFS_SUSPEND_SUSPENDED`.

SEE ALSO

`puffs`(3), `puffs_cc`(3)

BUGS

Currently the implementation works only for single-threaded file systems which do not use **puffs_cc**.

File system data and metadata are not always totally correctly synchronized at suspend. This will be fixed soon.

NAME

fputc, **putc**, **putchar**, **putc_unlocked**, **putchar_unlocked**, **putw** — output a character or word to a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

int
fputc(int c, FILE *stream);

int
putc(int c, FILE *stream);

int
putchar(int c);

int
putc_unlocked(int c, FILE *stream);

int
putchar_unlocked(int c);

int
putw(int w, FILE *stream);
```

DESCRIPTION

The **fputc**() function writes the character *c* (converted to an ''unsigned char'') to the output stream pointed to by *stream*.

putc() acts essentially identically to **fputc**(), but is a macro that expands in-line. It may evaluate *stream* more than once, so arguments given to **putc**() should not be expressions with potential side effects.

putchar() is identical to **putc**() with an output stream of *stdout*.

The **putc_unlocked**() and **putchar_unlocked**() functions provide functionality identical to that of **putc**() and **putchar**(), respectively, but do not perform implicit locking of the streams they operate on. In multi-threaded programs they may be used *only* within a scope in which the stream has been successfully locked by the calling thread using either flockfile(3) or ftrylockfile(3), and may later be released using funlockfile(3).

The **putw**() function writes the specified *int* to the named output *stream*.

RETURN VALUES

The functions, **fputc**(), **putc**() and **putchar**() return the character written. If an error occurs, the value EOF is returned. The **putw**() function returns 0 on success; EOF is returned if a write error occurs, or if an attempt is made to write a read-only stream.

SEE ALSO

ferror(3), fopen(3), getc(3), stdio(3)

STANDARDS

The functions **fputc**(), **putc**(), and **putchar**(), conform to ANSI X3.159-1989 (“ANSI C89”). The functions **putc_unlocked**() and **putchar_unlocked**() conform to ISO/IEC 9945-1:1996 (“POSIX.1”). A function **putw**() function appeared in Version 6 AT&T UNIX.

BUGS

The size and byte order of an *int* varies from one machine to another, and **putw**() is not recommended for portable applications.

NAME

fputwc, **putwc**, **putwchar**, — output a wide-character to a stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t
fputwc(wchar_t wc, FILE *stream);

wint_t
putwc(wchar_t wc, FILE *stream);

wint_t
putwchar(wchar_t wc);
```

DESCRIPTION

The **fputwc**() function writes the wide-character *wc* to the output stream pointed to by *stream*.

putwc() acts essentially identically to **fputwc**(), but is a macro that expands in-line. It may evaluate *stream* more than once, so arguments given to **putwc**() should not be expressions with potential side effects.

putwchar() is identical to **putwc**() with an output stream of *stdout*.

RETURN VALUES

The functions **fputwc**(), **putwc**(), and **putwchar**() return the wide-character written. If an error occurs, the value WEOF is returned.

SEE ALSO

ferror(3), fopen(3), getwc(3), stdio(3)

STANDARDS

The functions **fputwc**(), **putwc**(), and **putwchar**(), conform to ISO/IEC 9899:1999 (“ISO C99”).

NAME

pw_getconf — password encryption configuration access function

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

void
pw_getconf(char *data, size_t len, const char *key, const char *option);
```

DESCRIPTION

The **pw_getconf**() function reads `/etc/passwd.conf` and retrieves the value of the option specified by `option` from the section given by `key`. If no suitable entry is found for the `key` an empty string will be returned in data.

To retrieve default values the key `default` can be used. In this case, if `/etc/passwd.conf` does not exist or does not contain a `default` section, the built-in defaults will be returned. They are as follows:

option	**data**
ypcipher	old
localcipher	old

An empty string is returned for all errors.

FILES

`/etc/passwd.conf`

ERRORS

pw_getconf() will fail if:

`[ENOTDIR]` There is no key in `/etc/passwd.conf` named `key`.

`[ENOENT]` There is no option named `option` in the specified key.

SEE ALSO

`passwd`(5), `passwd.conf`(5)

HISTORY

The **pw_getconf**() function first appeared in NetBSD 1.6.

NAME

pw_init, **pw_edit**, **pw_prompt**, **pw_copy**, **pw_copyx**, **pw_scan**, **pw_error** — utility functions for interactive passwd file updates

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <pwd.h>
#include <util.h>

void
pw_init(void);

void
pw_edit(int notsetuid, const char *filename);

void
pw_prompt(void);

void
pw_copy(int ffd, int tfd, struct passwd *pw, struct passwd *old_pw);

int
pw_copyx(int ffd, int tfd, struct passwd *pw, struct passwd *old_pw,
    char *errbuf, size_t errbufsz);

int
pw_scan(char *bp, struct passwd *pw, int *flags);

void
pw_error(const char *name, int err, int eval);
```

DESCRIPTION

These functions are designed as conveniences for interactive programs which update the passwd file and do nothing else. They generally handle errors by printing out a message to the standard error stream and possibly aborting the process.

The **pw_init**() function prepares for a passwd update by unlimiting all resource constraints, disabling core dumps (thus preventing dumping the contents of the passwd database into a world-readable file), and disabling most signals.

The **pw_edit**() function runs an editor (named by the environment variable EDITOR, or `/usr/bin/vi` if EDITOR is not set) on the file *filename* (or `/etc/ptmp` if *filename* is NULL). If *notsetuid* is nonzero, **pw_edit**() will set the effective user and group ID to the real user and group ID before running the editor.

The **pw_prompt**() function asks the user whether he or she wants to re-edit the password file; if the answer is no, **pw_prompt**() deletes the lock file and exits the process.

The **pw_copy**() function reads a passwd file from *ffd* and writes it to *tfd*, updating the entry corresponding to pw->pw_name with the information in *pw*. If *old_pw* is not NULL, it checks to make sure the old entry is the same as the one described in *old_pw* or the process is aborted. If an entry is not found to match *pw*, a new entry is appended to the passwd file only if the real user ID is 0. If an error occurs, **pw_copy**() will display a message on `stderr` and call **pw_error**().

The **pw_copyx**() function performs the same operation as **pw_copy**() with the exception of error handling. Upon an error, **pw_copyx**() will write an error message into the buffer pointed to by *errbuf* which has the size *errbufsz*.

The **pw_scan**() function accepts in *bp* a passwd entry as it would be represented in /etc/master.passwd and fills in *pw* with corresponding values; string fields in *pw* will be pointers into *bp*. Some characters in *bp* will be overwritten with 0s in order to terminate the strings pointed to by *pw*. If *flags* is non-null, it should be cleared and the following options enabled if required:

_PASSWORD_NOWARN Don't print warnings.

_PASSWORD_OLDFMT Parse *bp* as an old format entry as found in /etc/passwd.

Upon return it is cleared, and filled in with the following flags:

_PASSWORD_NOUID The uid field of *bp* is empty.

_PASSWORD_NOGID The gid field of *bp* is empty.

_PASSWORD_NOCHG The change field of *bp* is empty.

_PASSWORD_NOEXP The expire field of *bp* is empty.

The **pw_error**() function displays an error message, aborts the current passwd update, and exits the current process. If *err* is non-zero, a warning message beginning with *name* is printed for the current value of *errno*. The process exits with status *eval*.

RETURN VALUES

The **pw_copyx**() function returns 1 if the new password entry was successfully written to the destination file, and 0 otherwise.

The **pw_scan**() function prints a warning message and returns 0 if the string in the *bp* argument is not a valid passwd string. Otherwise, **pw_scan**() returns 1.

FILES

/etc/master.passwd
/etc/ptmp

SEE ALSO

pw_lock(3), passwd(5)

NAME

pw_lock, **pw_mkdb**, **pw_abort**, **pw_setprefix**, **pw_getprefix** — passwd file update functions

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
pw_lock(int retries);

int
pw_mkdb(const char *username, int secureonly);

void
pw_abort(void);

int
pw_setprefix(const char *new_prefix);

const char *
pw_getprefix(void);
```

DESCRIPTION

The **pw_lock**(), **pw_mkdb**(), and **pw_abort**() functions allow a program to update the system passwd database.

The **pw_lock**() function attempts to lock the passwd database by creating the file /etc/ptmp, and returns the file descriptor of that file. If *retries* is greater than zero, **pw_lock**() will try multiple times to open /etc/ptmp, waiting one second between tries. In addition to being a lock file, /etc/ptmp will also hold the contents of the new passwd file.

The **pw_mkdb**() function updates the passwd file from the contents of /etc/ptmp. You should finish writing to and close the file descriptor returned by **pw_lock**() before calling **pw_mkdb**(). If **pw_mkdb**() fails and you do not wish to retry, you should make sure to call **pw_abort**() to clean up the lock file. If the *username* argument is not NULL, only database entries pertaining to the specified user will be modified. If the *secureonly* argument is non-zero, only the secure database will be updated.

The **pw_abort**() function aborts a passwd file update by deleting /etc/ptmp. The passwd database remains unchanged.

The **pw_setprefix**() function defines the root directory used for passwd file updates. If the prefix is set to /newroot **pw_lock**() will operate on /newroot/etc/ptmp afterwards. The default prefix is an empty string.

The **pw_getprefix**() function returns the root directory which is currently used for passwd file updates.

RETURN VALUES

The **pw_lock**() and **pw_mkdb**() functions return -1 if they are unable to complete properly.

FILES

/etc/master.passwd
/etc/ptmp

SEE ALSO

`pw_init`(3), `pwd_mkdb`(8)

NAME

pw_policy_load, **pw_policy_test** — password policy enforcement

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

pw_policy_t
pw_policy_load(void *key, int how);

int
pw_policy_test(pw_policy_t policy, char *pw);

void
pw_policy_free(pw_policy_t policy);
```

DESCRIPTION

The **pw_policy_load**(), **pw_policy_test**(), and **pw_policy_free**() functions are used as an interface to the system's password policy as specified in /etc/passwd.conf.

pw_policy_load() will load a password policy and return a pointer to a *pw_policy_t* containing it. It is the caller's responsibility to free this pointer using **pw_policy_free**().

Using pw_getconf(3) terminology, **pw_policy_load**() accepts a *key* to be used when searching /etc/passwd.conf for a password policy. This key contains various options describing different policies. Some built-in ones are described along with their syntax below.

To allow calling from various program contexts and using various password policy retrieval schemes, *how* tells **pw_policy_load**() how to treat *key*.

Possible values for *how* are:

PW_POLICY_BYSTRING
: *key* is used as a *char **, looking up the string it contains in /etc/passwd.conf.

PW_POLICY_BYPASSWD
: *key* is used as a *struct passwd **, first looking up the username in *pw_name*, and if no key can be found, it will try the login class in *pw_class*.

PW_POLICY_BYGROUP
: *key* is used as a *struct group **, looking up the group name in *gr_name*.

If *key* is NULL, or no specified key can be found, the default key, "pw_policy", is used. If even the default key can't be found, the password is accepted as no policy is defined.

pw_policy_test() can be used to check if the password in *pw* is compliant with the policy in *policy*.

BUILT-IN POLICY SYNTAX

Available built-in policy options include the following:

length	Length of the password.
uppercase	Number of upper-case characters in the password.
lowercase	Number of lower-case characters in the password.
digits	Number of digits in the password.

punctuation	Number of punctuation characters in the password.
nclasses	Number of different character classes in the password.
ntoggles	How often a user has to toggle between character classes in the password.

Options are used inside keys. An option uses a format of “option = value”. For the built-in options, we use either “N” or “N-M” for the value.

The first, “N” format, specifies a single length. For example, the following option specifies that the password should have exactly 3 upper-case characters:

```
uppercase = 3
```

The second, “N-M” format, can be used to specify a range. Forcing a policy for number of digits between 1 and 4 would be:

```
digits = 1-4
```

The characters ‘0’ and ‘∗’ can also be used to indicate “not allowed” and “any number”, respectively. To illustrate, the following example states that the number of punctuation characters should be at least two:

```
punctuation = 2-∗
```

No more than 7 digits:

```
digits = ∗-7
```

Any number of lower-case characters:

```
lowercase = ∗
```

Upper-case characters not allowed:

```
uppercase = 0
```

To specify that the password must be at least 8 characters long:

```
length = 8-∗
```

Specifying a password must have at least 3 different character classes:

```
nclasses = 3-∗
```

And that the user must change character class every 2 characters:

```
ntoggles = ∗-2
```

Note that when using the “nclasses” directive, the policy will be initialized to allow any number of characters from all classes. If desired, this should be overridden after the “nclasses” option.

RETURN VALUES

pw_policy_load() returns a *pw_policy_t* on success, or NULL on failure, in which case the *errno* variable will be set to any of the following values indicating the reason for the failure:

[ENOENT]	The /etc/passwd.conf is missing.
[EINVAL]	Invalid value for the *how* parameter or an invalid value in the password policy specification.

pw_policy_load() can also set *errno* to a value returned by the called handlers and/or malloc(3).

pw_policy_test() returns 0 if the password follows the policy, or −1 if it doesn’t, *errno* can be set to any of the following values:

[EPERM] The password does not follow the password policy.

[EINVAL] NULL pointer was passed as the password.

In addition, *errno* can be set to any error code returned by the handlers.

FILES

`/etc/passwd.conf` password configuration file.

EXAMPLES

Declare a password policy storage:

```
pw_policy_t policy;
```

Load the system global password policy into *policy*:

```
policy = pw_policy_load(NULL, 0);
if (policy == NULL)
        errx(1, "Can't load password policy");
```

Load a policy for a user whose password database entry is in *pw_entry* into *policy*:

```
policy = pw_policy_load(pw_entry, PW_POLICY_BYPASSWD);
if (policy == NULL)
        errx(1, "Can't load password policy for \"%s\"", pw_entry->pw_name);
```

Note that **pw_policy_load**() will first look for a password policy for the username in *pw_entry->pw_name*, if not found, it will try looking for a policy for the login class in *pw_entry->pw_class*, and if it can't find such either it will fallback to the default key, "pw_policy".

Load the password policy for a group whose group database entry is in *grent*, into *policy*:

```
policy = pw_policy_load(grent, PW_POLICY_BYGROUP);
if (policy == NULL)
        errx(1, "Can't load password policy for \"%s\"", grent->gr_name);
```

Check if *the_password* follows the policy in *policy*:

```
if (pw_policy_test(policy, the_password) != 0)
        warnx("Please refer to the password policy");
```

After finished using the password policy, free it:

```
pw_policy_free(policy);
```

An example for a common default password policy in `/etc/passwd.conf`:

```
pw_policy:
  length = 8-*          # At least 8 characters long,
  lowercase = 1-*       # combining lowercase,
  uppercase = 1-*       # uppercase,
  digits = 1-*          # and digits.
  punctuation = *       # Punctuation is optional.
```

A different policy that might be used:

```
  nclasses = 3-*        # At least 3 different character classes,
  ntoggles = *-2        # not more than 2 same class in a row.
```

SEE ALSO

`pw_getconf`(3), `passwd.conf`(5)

HISTORY

The **`pw_policy_load`**(), **`pw_policy_test`**(), and **`pw_policy_free`**() functions first appeared in NetBSD 4.0.

AUTHORS

Elad Efrat ⟨elad@NetBSD.org⟩

NAME

pwcache, **user_from_uid**, **group_from_gid** — cache password and group entries

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <pwd.h>

const char *
user_from_uid(uid_t uid, int nouser);

int
uid_from_user(const char *name, uid_t *uid);

int
pwcache_userdb(int (*setpassent)(int), void (*endpwent)(void),
    struct passwd * (*getpwnam)(const char *),
    struct passwd * (*getpwuid)(uid_t));

#include <grp.h>

const char *
group_from_gid(gid_t gid, int nogroup);

int
gid_from_group(const char *name, gid_t *gid);

int
pwcache_groupdb(int (*setgroupent)(int), void (*endgrent)(void),
    struct group * (*getgrnam)(const char *),
    struct group * (*getgrgid)(gid_t));
```

DESCRIPTION

The **user_from_uid**() function returns the user name associated with the argument *uid*. The user name is cached so that multiple calls with the same *uid* do not require additional calls to getpwuid(3). If there is no user associated with the *uid*, a pointer is returned to a string representation of the *uid*, unless the argument *nouser* is non-zero, in which case a NULL pointer is returned.

The **group_from_gid**() function returns the group name associated with the argument *gid*. The group name is cached so that multiple calls with the same *gid* do not require additional calls to getgrgid(3). If there is no group associated with the *gid*, a pointer is returned to a string representation of the *gid*, unless the argument *nogroup* is non-zero, in which case a NULL pointer is returned.

The **uid_from_user**() function returns the uid associated with the argument *name*. The uid is cached so that multiple calls with the same *name* do not require additional calls to getpwnam(3). If there is no uid associated with the *name*, the **uid_from_user**() function returns −1; otherwise it stores the uid at the location pointed to by *uid* and returns 0.

The **gid_from_group**() function returns the gid associated with the argument *name*. The gid is cached so that multiple calls with the same *name* do not require additional calls to getgrnam(3). If there is no gid associated with the *name*, the **gid_from_group**() function returns −1; otherwise it stores the gid at the location pointed to by *gid* and returns 0.

The **pwcache_userdb**() function changes the user database access routines which **user_from_uid**() and **uid_from_user**() call to search for users. The caches are flushed and the existing **endpwent**() method is called before switching to the new routines. *getpwnam* and *getpwuid* must be provided, and

setpassent and *endpwent* may be NULL pointers.

The **pwcache_groupdb**() function changes the group database access routines which **group_from_gid**() and **gid_from_group**() call to search for groups. The caches are flushed and the existing **endgrent**() method is called before switching to the new routines. *getgrnam* and *getgrgid* must be provided, and *setgroupent* and *endgrent* may be NULL pointers.

SEE ALSO

getgrgid(3), getgrnam(3), getpwnam(3), getpwuid(3)

HISTORY

The **user_from_uid**() and **group_from_gid**() functions first appeared in 4.4BSD.

The **uid_from_user**() and **gid_from_group**() functions first appeared in NetBSD 1.4.

The **pwcache_userdb**() and **pwcache_groupdb**() functions first appeared in NetBSD 1.6.

NAME

qabs — return the absolute value of a quad integer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

quad_t
qabs(quad_t j);
```

DESCRIPTION

The **qabs**() function returns the absolute value of the quad integer *j*.

SEE ALSO

abs(3), cabs(3), floor(3), labs(3), llabs(3), math(3)

BUGS

The absolute value of the most negative integer remains negative.

NAME

qdiv — return quotient and remainder from division

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

qdiv_t
qdiv(quad_t num, quad_t denom);
```

DESCRIPTION

The **qdiv**() function computes the value *num/denom* and returns the quotient and remainder in a structure named *qdiv_t* that contains two *quad integer* members named *quot* and *rem*.

SEE ALSO

div(3), ldiv(3), lldiv(3), math(3)

NAME

qsort, **heapsort**, **mergesort** — sort functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

void
qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));

int
heapsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));

int
mergesort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));
```

DESCRIPTION

The **qsort**() function is a modified partition-exchange sort, or quicksort. The **heapsort**() function is a modified selection sort. The **mergesort**() function is a modified merge sort with exponential search intended for sorting data with pre-existing order.

The **qsort**() and **heapsort**() functions sort an array of *nmemb* objects, the initial member of which is pointed to by *base*. The size of each object is specified by *size*. **mergesort**() behaves similarly, but *requires* that *size* be greater than "sizeof(void *) / 2".

The contents of the array *base* are sorted in ascending order according to a comparison function pointed to by *compar*, which requires two arguments pointing to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

The functions **qsort**() and **heapsort**() are *not* stable, that is, if two members compare as equal, their order in the sorted array is undefined. The function **mergesort**() is stable.

The **qsort**() function is an implementation of C.A.R. Hoare's ''quicksort'' algorithm, a variant of partition-exchange sorting; in particular, see D.E. Knuth's Algorithm Q. **qsort**() takes O N lg N average time. This implementation uses median selection to avoid its O N**2 worst-case behavior.

The **heapsort**() function is an implementation of J.W.J. William's ''heapsort'' algorithm, a variant of selection sorting; in particular, see D.E. Knuth's Algorithm H. **heapsort**() takes O N lg N worst-case time. Its *only* advantage over **qsort**() is that it uses almost no additional memory; while **qsort**() does not allocate memory, it is implemented using recursion.

The function **mergesort**() requires additional memory of size *nmemb* * *size* bytes; it should be used only when space is not at a premium. **mergesort**() is optimized for data with pre-existing order; its worst case time is O N lg N; its best case is O N.

Normally, **qsort**() is faster than **mergesort**() is faster than **heapsort**(). Memory availability and pre-existing order in the data can make this untrue.

RETURN VALUES

The **qsort**() function returns no value.

Upon successful completion, **heapsort**() and **mergesort**() return 0. Otherwise, they return −1 and the global variable *errno* is set to indicate the error.

ERRORS

The **heapsort**() function succeeds unless:

[EINVAL] The *size* argument is zero, or, the *size* argument to **mergesort**() is less than "sizeof(void ∗) / 2".

[ENOMEM] **heapsort**() or **mergesort**() were unable to allocate memory.

COMPATIBILITY

Previous versions of **qsort**() did not permit the comparison routine itself to call **qsort**(). This is no longer true.

SEE ALSO

sort(1), radixsort(3)

Hoare, C.A.R., "Quicksort", *The Computer Journal*, 5:1, pp. 10-15, 1962.

Williams, J.W.J, "Heapsort", *Communications of the ACM*, 7:1, pp. 347-348, 1964.

Knuth, D.E., "Sorting and Searching", *The Art of Computer Programming*, Vol. 3, pp. 114-123, 145-149, 1968.

McIlroy, P.M., "Optimistic Sorting and Information Theoretic Complexity", *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 467-474, 1993.

Bentley, J.L. and McIlroy, M.D., "Engineering a Sort Function", *Software-Practice and Experience*, Vol. 23, pp. 1249-1265, 1993.

STANDARDS

The **qsort**() function conforms to ANSI X3.159-1989 ("ANSI C89").

NAME

SLIST_HEAD, **SLIST_HEAD_INITIALIZER**, **SLIST_ENTRY**, **SLIST_INIT**, **SLIST_INSERT_AFTER**, **SLIST_INSERT_HEAD**, **SLIST_REMOVE_HEAD**, **SLIST_REMOVE**, **SLIST_FOREACH**, **SLIST_EMPTY**, **SLIST_FIRST**, **SLIST_NEXT**, **SIMPLEQ_HEAD**, **SIMPLEQ_HEAD_INITIALIZER**, **SIMPLEQ_ENTRY**, **SIMPLEQ_INIT**, **SIMPLEQ_INSERT_HEAD**, **SIMPLEQ_INSERT_TAIL**, **SIMPLEQ_INSERT_AFTER**, **SIMPLEQ_REMOVE_HEAD**, **SIMPLEQ_REMOVE**, **SIMPLEQ_FOREACH**, **SIMPLEQ_EMPTY**, **SIMPLEQ_FIRST**, **SIMPLEQ_NEXT**, **STAILQ_HEAD**, **STAILQ_HEAD_INITIALIZER**, **STAILQ_ENTRY**, **STAILQ_INIT**, **STAILQ_INSERT_HEAD**, **STAILQ_INSERT_TAIL**, **STAILQ_INSERT_AFTER**, **STAILQ_REMOVE_HEAD**, **STAILQ_REMOVE**, **STAILQ_FOREACH**, **STAILQ_EMPTY**, **STAILQ_FIRST**, **STAILQ_NEXT**, **STAILQ_CONCAT**, **LIST_HEAD**, **LIST_HEAD_INITIALIZER**, **LIST_ENTRY**, **LIST_INIT**, **LIST_INSERT_AFTER**, **LIST_INSERT_BEFORE**, **LIST_INSERT_HEAD**, **LIST_REMOVE**, **LIST_FOREACH**, **LIST_EMPTY**, **LIST_FIRST**, **LIST_NEXT**, **TAILQ_HEAD**, **TAILQ_HEAD_INITIALIZER**, **TAILQ_ENTRY**, **TAILQ_INIT**, **TAILQ_INSERT_HEAD**, **TAILQ_INSERT_TAIL**, **TAILQ_INSERT_AFTER**, **TAILQ_INSERT_BEFORE**, **TAILQ_REMOVE**, **TAILQ_FOREACH**, **TAILQ_FOREACH_REVERSE**, **TAILQ_EMPTY**, **TAILQ_FIRST**, **TAILQ_NEXT**, **TAILQ_LAST**, **TAILQ_PREV**, **TAILQ_CONCAT**, **CIRCLEQ_HEAD**, **CIRCLEQ_HEAD_INITIALIZER**, **CIRCLEQ_ENTRY**, **CIRCLEQ_INIT**, **CIRCLEQ_INSERT_AFTER**, **CIRCLEQ_INSERT_BEFORE**, **CIRCLEQ_INSERT_HEAD**, **CIRCLEQ_INSERT_TAIL**, **CIRCLEQ_REMOVE**, **CIRCLEQ_FOREACH**, **CIRCLEQ_FOREACH_REVERSE**, **CIRCLEQ_EMPTY**, **CIRCLEQ_FIRST**, **CIRCLEQ_LAST**, **CIRCLEQ_NEXT**, **CIRCLEQ_PREV**, **CIRCLEQ_LOOP_NEXT**, **CIRCLEQ_LOOP_PREV** — implementations of singly-linked lists, simple queues, lists, tail queues, and circular queues

SYNOPSIS

```
#include <sys/queue.h>

SLIST_HEAD(HEADNAME, TYPE);

SLIST_HEAD_INITIALIZER(head);

SLIST_ENTRY(TYPE);

SLIST_INIT(SLIST_HEAD *head);

SLIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, SLIST_ENTRY NAME);

SLIST_INSERT_HEAD(SLIST_HEAD *head, TYPE *elm, SLIST_ENTRY NAME);

SLIST_REMOVE_HEAD(SLIST_HEAD *head, SLIST_ENTRY NAME);

SLIST_REMOVE(SLIST_HEAD *head, TYPE *elm, TYPE, SLIST_ENTRY NAME);

SLIST_FOREACH(TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME);

int
SLIST_EMPTY(SLIST_HEAD *head);

TYPE *
SLIST_FIRST(SLIST_HEAD *head);

TYPE *
SLIST_NEXT(TYPE *elm, SLIST_ENTRY NAME);

SIMPLEQ_HEAD(HEADNAME, TYPE);
```

SIMPLEQ_HEAD_INITIALIZER(*head*);

SIMPLEQ_ENTRY(*TYPE*);

SIMPLEQ_INIT(*SIMPLEQ_HEAD *head*);

SIMPLEQ_INSERT_HEAD(*SIMPLEQ_HEAD *head*, *TYPE *elm*, *SIMPLEQ_ENTRY NAME*);

SIMPLEQ_INSERT_TAIL(*SIMPLEQ_HEAD *head*, *TYPE *elm*, *SIMPLEQ_ENTRY NAME*);

SIMPLEQ_INSERT_AFTER(*SIMPLEQ_HEAD *head*, *TYPE *listelm*, *TYPE *elm*, *SIMPLEQ_ENTRY NAME*);

SIMPLEQ_REMOVE_HEAD(*SIMPLEQ_HEAD *head*, *SIMPLEQ_ENTRY NAME*);

SIMPLEQ_REMOVE(*SIMPLEQ_HEAD *head*, *TYPE *elm*, *TYPE*, *SIMPLEQ_ENTRY NAME*);

SIMPLEQ_FOREACH(*TYPE *var*, *SIMPLEQ_HEAD *head*, *SIMPLEQ_ENTRY NAME*);

int
SIMPLEQ_EMPTY(*SIMPLEQ_HEAD *head*);

*TYPE **
SIMPLEQ_FIRST(*SIMPLEQ_HEAD *head*);

*TYPE **
SIMPLEQ_NEXT(*TYPE *elm*, *SIMPLEQ_ENTRY NAME*);

STAILQ_HEAD(*HEADNAME*, *TYPE*);

STAILQ_HEAD_INITIALIZER(*head*);

STAILQ_ENTRY(*TYPE*);

STAILQ_INIT(*STAILQ_HEAD *head*);

STAILQ_INSERT_HEAD(*STAILQ_HEAD *head*, *TYPE *elm*, *STAILQ_ENTRY NAME*);

STAILQ_INSERT_TAIL(*STAILQ_HEAD *head*, *TYPE *elm*, *STAILQ_ENTRY NAME*);

STAILQ_INSERT_AFTER(*STAILQ_HEAD *head*, *TYPE *listelm*, *TYPE *elm*, *STAILQ_ENTRY NAME*);

STAILQ_REMOVE_HEAD(*STAILQ_HEAD *head*, *STAILQ_ENTRY NAME*);

STAILQ_REMOVE(*STAILQ_HEAD *head*, *TYPE *elm*, *TYPE*, *STAILQ_ENTRY NAME*);

STAILQ_FOREACH(*TYPE *var*, *STAILQ_HEAD *head*, *STAILQ_ENTRY NAME*);

int
STAILQ_EMPTY(*STAILQ_HEAD *head*);

*TYPE **
STAILQ_FIRST(*STAILQ_HEAD *head*);

*TYPE **
STAILQ_NEXT(*TYPE *elm*, *STAILQ_ENTRY NAME*);

STAILQ_CONCAT(*STAILQ_HEAD *head1*, *STAILQ_HEAD *head2*);

LIST_HEAD(*HEADNAME*, *TYPE*);

LIST_HEAD_INITIALIZER(*head*);

LIST_ENTRY(*TYPE*);

LIST_INIT(*LIST_HEAD *head*);

LIST_INSERT_AFTER(*TYPE *listelm*, *TYPE *elm*, *LIST_ENTRY NAME*);

LIST_INSERT_BEFORE(*TYPE *listelm*, *TYPE *elm*, *LIST_ENTRY NAME*);

LIST_INSERT_HEAD(*LIST_HEAD *head*, *TYPE *elm*, *LIST_ENTRY NAME*);

LIST_REMOVE(*TYPE *elm*, *LIST_ENTRY NAME*);

LIST_FOREACH(*TYPE *var*, *LIST_HEAD *head*, *LIST_ENTRY NAME*);

int
LIST_EMPTY(*LIST_HEAD *head*);

*TYPE **
LIST_FIRST(*LIST_HEAD *head*);

*TYPE **
LIST_NEXT(*TYPE *elm*, *LIST_ENTRY NAME*);

TAILQ_HEAD(*HEADNAME*, *TYPE*);

TAILQ_HEAD_INITIALIZER(*head*);

TAILQ_ENTRY(*TYPE*);

TAILQ_INIT(*TAILQ_HEAD *head*);

TAILQ_INSERT_HEAD(*TAILQ_HEAD *head*, *TYPE *elm*, *TAILQ_ENTRY NAME*);

TAILQ_INSERT_TAIL(*TAILQ_HEAD *head*, *TYPE *elm*, *TAILQ_ENTRY NAME*);

TAILQ_INSERT_AFTER(*TAILQ_HEAD *head*, *TYPE *listelm*, *TYPE *elm*, *TAILQ_ENTRY NAME*);

TAILQ_INSERT_BEFORE(*TYPE *listelm*, *TYPE *elm*, *TAILQ_ENTRY NAME*);

TAILQ_REMOVE(*TAILQ_HEAD *head*, *TYPE *elm*, *TAILQ_ENTRY NAME*);

TAILQ_FOREACH(*TYPE *var*, *TAILQ_HEAD *head*, *TAILQ_ENTRY NAME*);

TAILQ_FOREACH_REVERSE(*TYPE *var*, *TAILQ_HEAD *head*, *HEADNAME*, *TAILQ_ENTRY NAME*);

int
TAILQ_EMPTY(*TAILQ_HEAD *head*);

*TYPE **
TAILQ_FIRST(*TAILQ_HEAD *head*);

*TYPE **
TAILQ_NEXT(*TYPE *elm*, *TAILQ_ENTRY NAME*);

*TYPE **
TAILQ_LAST(*TAILQ_HEAD *head*, *HEADNAME*);

*TYPE **
TAILQ_PREV(*TYPE *elm*, *HEADNAME*, *TAILQ_ENTRY NAME*);

TAILQ_CONCAT(*TAILQ_HEAD *head1*, *TAILQ_HEAD *head2*, *TAILQ_ENTRY NAME*);

CIRCLEQ_HEAD(*HEADNAME*, *TYPE*);

CIRCLEQ_HEAD_INITIALIZER(*head*);

CIRCLEQ_ENTRY(*TYPE*);

CIRCLEQ_INIT(*CIRCLEQ_HEAD *head*);

CIRCLEQ_INSERT_AFTER(*CIRCLEQ_HEAD *head*, *TYPE *listelm*, *TYPE *elm*, *CIRCLEQ_ENTRY NAME*);

CIRCLEQ_INSERT_BEFORE(*CIRCLEQ_HEAD *head*, *TYPE *listelm*, *TYPE *elm*, *CIRCLEQ_ENTRY NAME*);

CIRCLEQ_INSERT_HEAD(*CIRCLEQ_HEAD *head*, *TYPE *elm*, *CIRCLEQ_ENTRY NAME*);

CIRCLEQ_INSERT_TAIL(*CIRCLEQ_HEAD *head*, *TYPE *elm*, *CIRCLEQ_ENTRY NAME*);

CIRCLEQ_REMOVE(*CIRCLEQ_HEAD *head*, *TYPE *elm*, *CIRCLEQ_ENTRY NAME*);

CIRCLEQ_FOREACH(*TYPE *var*, *CIRCLEQ_HEAD *head*, *CIRCLEQ_ENTRY NAME*);

CIRCLEQ_FOREACH_REVERSE(*TYPE *var*, *CIRCLEQ_HEAD *head*, *CIRCLEQ_ENTRY NAME*);

int
CIRCLEQ_EMPTY(*CIRCLEQ_HEAD *head*);

*TYPE **
CIRCLEQ_FIRST(*CIRCLEQ_HEAD *head*);

*TYPE **
CIRCLEQ_LAST(*CIRCLEQ_HEAD *head*);

*TYPE **
CIRCLEQ_NEXT(*TYPE *elm*, *CIRCLEQ_ENTRY NAME*);

*TYPE **
CIRCLEQ_PREV(*TYPE *elm*, *CIRCLEQ_ENTRY NAME*);

*TYPE **
CIRCLEQ_LOOP_NEXT(*CIRCLEQ_HEAD *head*, *TYPE *elm*, *CIRCLEQ_ENTRY NAME*);

*TYPE **
CIRCLEQ_LOOP_PREV(*CIRCLEQ_HEAD *head*, *TYPE *elm*, *CIRCLEQ_ENTRY NAME*);

DESCRIPTION

These macros define and operate on five types of data structures: singly-linked lists, simple queues, lists, tail queues, and circular queues. All five structures support the following functionality:

1. Insertion of a new entry at the head of the list.
2. Insertion of a new entry before or after any element in the list.
3. Removal of any entry in the list.
4. Forward traversal through the list.

Singly-linked lists are the simplest of the five data structures and support only the above functionality. Singly-linked lists are ideal for applications with large datasets and few or no removals, or for implementing a LIFO queue.

Simple queues add the following functionality:

1. Entries can be added at the end of a list.
2. They may be concatenated.

However:

1. Entries may not be added before any element in the list.
2. All list insertions and removals must specify the head of the list.
3. Each head entry requires two pointers rather than one.

Simple queues are ideal for applications with large datasets and few or no removals, or for implementing a FIFO queue.

All doubly linked types of data structures (lists, tail queues, and circle queues) additionally allow:

1. Insertion of a new entry before any element in the list.
2. O(1) removal of any entry in the list.

However:

1. Each element requires two pointers rather than one.
2. Code size and execution time of operations (except for removal) is about twice that of the singly-linked data-structures.

Linked lists are the simplest of the doubly linked data structures and support only the above functionality over singly-linked lists.

Tail queues add the following functionality:

1. Entries can be added at the end of a list.
2. They may be concatenated.

However:

1. All list insertions and removals, except insertion before another element, must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. Code size is about 15% greater and operations run about 20% slower than lists.

Circular queues add the following functionality:

1. Entries can be added at the end of a list.
2. They may be traversed backwards, from tail to head.

However:

1. All list insertions and removals must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. The termination condition for traversal is more complex.
4. Code size is about 40% greater and operations run about 45% slower than lists.

In the macro definitions, *TYPE* is the name of a user defined structure, that must contain a field of type `LIST_ENTRY`, `SIMPLEQ_ENTRY`, `SLIST_ENTRY`, `TAILQ_ENTRY`, or `CIRCLEQ_ENTRY`, named *NAME*. The argument *HEADNAME* is the name of a user defined structure that must be declared using the macros `LIST_HEAD`, `SIMPLEQ_HEAD`, `SLIST_HEAD`, `TAILQ_HEAD`, or `CIRCLEQ_HEAD`. See the examples below for further explanation of how these macros are used.

Summary of Operations

The following table summarizes the supported macros for each type of data structure.

	SLIST	LIST	SIMPLEQ	STAILQ	TAILQ	CIRCLEQ
_EMPTY	+	+	+	+	+	+
_FIRST	+	+	+	+	+	+
_FOREACH	+	+	+	+	+	+
_FOREACH_REVERSE	-	-	-	-	+	+
_INSERT_AFTER	+	+	+	+	+	+
_INSERT_BEFORE	-	+	-	-	+	+
_INSERT_HEAD	+	+	+	+	+	+
_INSERT_TAIL	-	-	+	+	+	+
_LAST	-	-	-	-	+	+
_LOOP_NEXT	-	-	-	-	-	+
_LOOP_PREV	-	-	-	-	-	+
_NEXT	+	+	+	+	+	+
_PREV	-	-	-	-	+	+
_REMOVE	+	+	+	+	+	+
_REMOVE_HEAD	+	-	+	+	-	-
_CONCAT	-	-	-	+	+	-

SINGLY-LINKED LISTS

A singly-linked list is headed by a structure defined by the **SLIST_HEAD** macro. This structure contains a single pointer to the first element on the list. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of O(n) removal for arbitrary elements. New elements can be added to the list after an existing element or at the head of the list. An *SLIST_HEAD* structure is declared as follows:

```
SLIST_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

```
struct HEADNAME *headp;
```

(The names `head` and `headp` are user selectable.)

The macro **SLIST_HEAD_INITIALIZER** evaluates to an initializer for the list *head*.

The macro **SLIST_EMPTY** evaluates to true if there are no elements in the list.

The macro **SLIST_ENTRY** declares a structure that connects the elements in the list.

The macro **SLIST_FIRST** returns the first element in the list or NULL if the list is empty.

The macro **SLIST_FOREACH** traverses the list referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro **SLIST_INIT** initializes the list referenced by *head*.

The macro **SLIST_INSERT_HEAD** inserts the new element *elm* at the head of the list.

The macro **SLIST_INSERT_AFTER** inserts the new element *elm* after the element *listelm*.

The macro **SLIST_NEXT** returns the next element in the list.

The macro **SLIST_REMOVE** removes the element *elm* from the list.

The macro **SLIST_REMOVE_HEAD** removes the first element from the head of the list. For optimum efficiency, elements being removed from the head of the list should explicitly use this macro instead of the generic **SLIST_REMOVE** macro.

SINGLY-LINKED LIST EXAMPLE

```
SLIST_HEAD(slisthead, entry) head =
    SLIST_HEAD_INITIALIZER(head);
struct slisthead *headp;                /* Singly-linked List head. */
struct entry {
        ...
        SLIST_ENTRY(entry) entries;     /* Singly-linked List. */
        ...
} *n1, *n2, *n3, *np;

SLIST_INIT(&head);                      /* Initialize the list. */

n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
SLIST_INSERT_HEAD(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
SLIST_INSERT_AFTER(n1, n2, entries);

SLIST_REMOVE(&head, n2, entry, entries);/* Deletion. */
free(n2);

n3 = SLIST_FIRST(&head);
SLIST_REMOVE_HEAD(&head, entries);      /* Deletion from the head. */
free(n3);
                                        /* Forward traversal. */
SLIST_FOREACH(np, &head, entries)
        np-> ...

while (!SLIST_EMPTY(&head)) {           /* List Deletion. */
        n1 = SLIST_FIRST(&head);
        SLIST_REMOVE_HEAD(&head, entries);
        free(n1);
}
```

SIMPLE QUEUES

A simple queue is headed by a structure defined by the **SIMPLEQ_HEAD** macro. This structure contains a pair of pointers, one to the first element in the simple queue and the other to the last element in the simple queue. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of O(n) removal for arbitrary elements. New elements can be added to the queue after an existing element, at the head of the queue, or at the end of the queue. A *SIMPLEQ_HEAD* structure is declared as follows:

```
SIMPLEQ_HEAD(HEADNAME, TYPE) head;
```

where HEADNAME is the name of the structure to be defined, and TYPE is the type of the elements to be linked into the simple queue. A pointer to the head of the simple queue can later be declared as:

```
struct HEADNAME *headp;
```

(The names head and headp are user selectable.)

The macro **SIMPLEQ_ENTRY** declares a structure that connects the elements in the simple queue.

The macro **SIMPLEQ_HEAD_INITIALIZER** provides a value which can be used to initialize a simple queue head at compile time, and is used at the point that the simple queue head variable is declared, like:

```
struct HEADNAME head = SIMPLEQ_HEAD_INITIALIZER(head);
```

The macro **SIMPLEQ_INIT** initializes the simple queue referenced by *head*.

The macro **SIMPLEQ_INSERT_HEAD** inserts the new element *elm* at the head of the simple queue.

The macro **SIMPLEQ_INSERT_TAIL** inserts the new element *elm* at the end of the simple queue.

The macro **SIMPLEQ_INSERT_AFTER** inserts the new element *elm* after the element *listelm*.

The macro **SIMPLEQ_REMOVE** removes *elm* from the simple queue.

The macro **SIMPLEQ_REMOVE_HEAD** removes the first element from the head of the simple queue. For optimum efficiency, elements being removed from the head of the queue should explicitly use this macro instead of the generic **SIMPLQ_REMOVE** macro.

The macro **SIMPLEQ_EMPTY** return true if the simple queue *head* has no elements.

The macro **SIMPLEQ_FIRST** returns the first element of the simple queue *head*.

The macro **SIMPLEQ_FOREACH** traverses the tail queue referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro **SIMPLEQ_NEXT** returns the element after the element *elm*.

The macros prefixed with "**STAILQ_**" (**STAILQ_HEAD**, **STAILQ_HEAD_INITIALIZER**, **STAILQ_ENTRY**, **STAILQ_INIT**, **STAILQ_INSERT_HEAD**, **STAILQ_INSERT_TAIL**, **STAILQ_INSERT_AFTER**, **STAILQ_REMOVE_HEAD**, **STAILQ_REMOVE**, **STAILQ_FOREACH**, **STAILQ_EMPTY**, **STAILQ_FIRST**, and **STAILQ_NEXT**) are functionally identical to these simple queue functions, and are provided for compatibility with FreeBSD.

SIMPLE QUEUE EXAMPLE

```
SIMPLEQ_HEAD(simplehead, entry) head;
struct simplehead *headp;             /* Simple queue head. */
struct entry {
        ...
        SIMPLEQ_ENTRY(entry) entries; /* Simple queue. */
        ...
} *n1, *n2, *np;

SIMPLEQ_INIT(&head);                  /* Initialize the queue. */

n1 = malloc(sizeof(struct entry));    /* Insert at the head. */
SIMPLEQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry));    /* Insert at the tail. */
SIMPLEQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry));    /* Insert after. */
SIMPLEQ_INSERT_AFTER(&head, n1, n2, entries);
                                      /* Forward traversal. */
SIMPLEQ_FOREACH(np, &head, entries)
        np-> ...
                                      /* Delete. */
while (SIMPLEQ_FIRST(&head) != NULL)
```

```
        SIMPLEQ_REMOVE_HEAD(&head, entries);
if (SIMPLEQ_EMPTY(&head))            /* Test for emptiness. */
        printf("nothing to do\n");
```

LISTS

A list is headed by a structure defined by the **LIST_HEAD** macro. This structure contains a single pointer to the first element on the list. The elements are doubly linked so that an arbitrary element can be removed without traversing the list. New elements can be added to the list after an existing element, before an existing element, or at the head of the list. A *LIST_HEAD* structure is declared as follows:

```
LIST_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

```
struct HEADNAME *headp;
```

(The names `head` and `headp` are user selectable.)

The macro **LIST_ENTRY** declares a structure that connects the elements in the list.

The macro **LIST_HEAD_INITIALIZER** provides a value which can be used to initialize a list head at compile time, and is used at the point that the list head variable is declared, like:

```
struct HEADNAME head = LIST_HEAD_INITIALIZER(head);
```

The macro **LIST_INIT** initializes the list referenced by *head*.

The macro **LIST_INSERT_HEAD** inserts the new element *elm* at the head of the list.

The macro **LIST_INSERT_AFTER** inserts the new element *elm* after the element *listelm*.

The macro **LIST_INSERT_BEFORE** inserts the new element *elm* before the element *listelm*.

The macro **LIST_REMOVE** removes the element *elm* from the list.

The macro **LIST_EMPTY** return true if the list *head* has no elements.

The macro **LIST_FIRST** returns the first element of the list *head*.

The macro **LIST_FOREACH** traverses the list referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro **LIST_NEXT** returns the element after the element *elm*.

LIST EXAMPLE

```
LIST_HEAD(listhead, entry) head;
struct listhead *headp;              /* List head. */
struct entry {
        ...
        LIST_ENTRY(entry) entries;   /* List. */
        ...
} *n1, *n2, *np;

LIST_INIT(&head);                    /* Initialize the list. */

n1 = malloc(sizeof(struct entry));   /* Insert at the head. */
LIST_INSERT_HEAD(&head, n1, entries);
```

```
n2 = malloc(sizeof(struct entry));    /* Insert after. */
LIST_INSERT_AFTER(n1, n2, entries);

n2 = malloc(sizeof(struct entry));    /* Insert before. */
LIST_INSERT_BEFORE(n1, n2, entries);
                                      /* Forward traversal. */
LIST_FOREACH(np, &head, entries)
        np-> ...
                                      /* Delete. */
while (LIST_FIRST(&head) != NULL)
        LIST_REMOVE(LIST_FIRST(&head), entries);
if (LIST_EMPTY(&head))                /* Test for emptiness. */
        printf("nothing to do\n");
```

TAIL QUEUES

A tail queue is headed by a structure defined by the **TAILQ_HEAD** macro. This structure contains a pair of pointers, one to the first element in the tail queue and the other to the last element in the tail queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the tail queue. New elements can be added to the queue after an existing element, before an existing element, at the head of the queue, or at the end the queue. A *TAILQ_HEAD* structure is declared as follows:

```
TAILQ_HEAD(HEADNAME, TYPE) head;
```

where HEADNAME is the name of the structure to be defined, and TYPE is the type of the elements to be linked into the tail queue. A pointer to the head of the tail queue can later be declared as:

```
struct HEADNAME *headp;
```

(The names head and headp are user selectable.)

The macro **TAILQ_ENTRY** declares a structure that connects the elements in the tail queue.

The macro **TAILQ_HEAD_INITIALIZER** provides a value which can be used to initialize a tail queue head at compile time, and is used at the point that the tail queue head variable is declared, like:

```
struct HEADNAME head = TAILQ_HEAD_INITIALIZER(head);
```

The macro **TAILQ_INIT** initializes the tail queue referenced by *head*.

The macro **TAILQ_INSERT_HEAD** inserts the new element *elm* at the head of the tail queue.

The macro **TAILQ_INSERT_TAIL** inserts the new element *elm* at the end of the tail queue.

The macro **TAILQ_INSERT_AFTER** inserts the new element *elm* after the element *listelm*.

The macro **TAILQ_INSERT_BEFORE** inserts the new element *elm* before the element *listelm*.

The macro **TAILQ_REMOVE** removes the element *elm* from the tail queue.

The macro **TAILQ_EMPTY** return true if the tail queue *head* has no elements.

The macro **TAILQ_FIRST** returns the first element of the tail queue *head*.

The macro **TAILQ_FOREACH** traverses the tail queue referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro **TAILQ_FOREACH_REVERSE** traverses the tail queue referenced by *head* in the reverse direction, assigning each element in turn to *var*.

The macro **TAILQ_NEXT** returns the element after the element *elm*.

The macro **TAILQ_CONCAT** concatenates the tail queue headed by *head2* onto the end of the one headed by *head1* removing all entries from the former.

TAIL QUEUE EXAMPLE

```
TAILQ_HEAD(tailhead, entry) head;
struct tailhead *headp;                 /* Tail queue head. */
struct entry {
        ...
        TAILQ_ENTRY(entry) entries;     /* Tail queue. */
        ...
} *n1, *n2, *np;

TAILQ_INIT(&head);                      /* Initialize the queue. */

n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
TAILQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry));      /* Insert at the tail. */
TAILQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
TAILQ_INSERT_AFTER(&head, n1, n2, entries);

n2 = malloc(sizeof(struct entry));      /* Insert before. */
TAILQ_INSERT_BEFORE(n1, n2, entries);
                                        /* Forward traversal. */
TAILQ_FOREACH(np, &head, entries)
        np-> ...
                                        /* Reverse traversal. */
TAILQ_FOREACH_REVERSE(np, &head, tailhead, entries)
        np-> ...
                                        /* Delete. */
while (TAILQ_FIRST(&head) != NULL)
        TAILQ_REMOVE(&head, TAILQ_FIRST(&head), entries);
if (TAILQ_EMPTY(&head))                       /* Test for emptiness. */
        printf("nothing to do\n");
```

CIRCULAR QUEUES

A circular queue is headed by a structure defined by the **CIRCLEQ_HEAD** macro. This structure contains a pair of pointers, one to the first element in the circular queue and the other to the last element in the circular queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the queue. New elements can be added to the queue after an existing element, before an existing element, at the head of the queue, or at the end of the queue. A *CIRCLEQ_HEAD* structure is declared as follows:

```
CIRCLEQ_HEAD(HEADNAME, TYPE) head;
```

where HEADNAME is the name of the structure to be defined, and TYPE is the type of the elements to be linked into the circular queue. A pointer to the head of the circular queue can later be declared as:

```
struct HEADNAME *headp;
```

(The names `head` and `headp` are user selectable.)

The macro **CIRCLEQ_ENTRY** declares a structure that connects the elements in the circular queue.

The macro **CIRCLEQ_HEAD_INITIALIZER** provides a value which can be used to initialize a circular queue head at compile time, and is used at the point that the circular queue head variable is declared, like:

```
struct HEADNAME head = CIRCLEQ_HEAD_INITIALIZER(head);
```

The macro **CIRCLEQ_INIT** initializes the circular queue referenced by *head*.

The macro **CIRCLEQ_INSERT_HEAD** inserts the new element *elm* at the head of the circular queue.

The macro **CIRCLEQ_INSERT_TAIL** inserts the new element *elm* at the end of the circular queue.

The macro **CIRCLEQ_INSERT_AFTER** inserts the new element *elm* after the element *listelm*.

The macro **CIRCLEQ_INSERT_BEFORE** inserts the new element *elm* before the element *listelm*.

The macro **CIRCLEQ_REMOVE** removes the element *elm* from the circular queue.

The macro **CIRCLEQ_EMPTY** return true if the circular queue *head* has no elements.

The macro **CIRCLEQ_FIRST** returns the first element of the circular queue *head*.

The macro **CIRCLEQ_FOREACH** traverses the circle queue referenced by *head* in the forward direction, assigning each element in turn to *var*. Each element is assigned exactly once.

The macro **CIRCLEQ_FOREACH_REVERSE** traverses the circle queue referenced by *head* in the reverse direction, assigning each element in turn to *var*. Each element is assigned exactly once.

The macro **CIRCLEQ_LAST** returns the last element of the circular queue *head*.

The macro **CIRCLEQ_NEXT** returns the element after the element *elm*.

The macro **CIRCLEQ_PREV** returns the element before the element *elm*.

The macro **CIRCLEQ_LOOP_NEXT** returns the element after the element *elm*. If *elm* was the last element in the queue, the first element is returned.

The macro **CIRCLEQ_LOOP_PREV** returns the element before the element *elm*. If *elm* was the first element in the queue, the last element is returned.

CIRCULAR QUEUE EXAMPLE

```
CIRCLEQ_HEAD(circleq, entry) head;
struct circleq *headp;                /* Circular queue head. */
struct entry {
        ...
        CIRCLEQ_ENTRY(entry) entries; /* Circular queue. */
        ...
} *n1, *n2, *np;

CIRCLEQ_INIT(&head);                  /* Initialize the circular queue. */

n1 = malloc(sizeof(struct entry));    /* Insert at the head. */
CIRCLEQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry));    /* Insert at the tail. */
CIRCLEQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry));    /* Insert after. */
```

```
CIRCLEQ_INSERT_AFTER(&head, n1, n2, entries);

n2 = malloc(sizeof(struct entry));    /* Insert before. */
CIRCLEQ_INSERT_BEFORE(&head, n1, n2, entries);
                                      /* Forward traversal. */
CIRCLEQ_FOREACH(np, &head, entries)
        np-> ...
                                      /* Reverse traversal. */
CIRCLEQ_FOREACH_REVERSE(np, &head, entries)
        np-> ...
                                      /* Delete. */
while (CIRCLEQ_FIRST(&head) != (void *)&head)
        CIRCLEQ_REMOVE(&head, CIRCLEQ_FIRST(&head), entries);
if (CIRCLEQ_EMPTY(&head))             /* Test for emptiness. */
        printf("nothing to do\n");
```

HISTORY

The **queue** functions first appeared in 4.4BSD. The **SIMPLEQ** functions first appeared in NetBSD 1.2. The **SLIST** and **STAILQ** functions first appeared in FreeBSD 2.1.5. The **CIRCLEQ_LOOP** functions first appeared in NetBSD 4.0.

NAME

radixsort, **sradixsort** — radix sort

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <limits.h>
#include <stdlib.h>

int
radixsort(const u_char **base, int nmemb, u_char *table, u_int endbyte);

int
sradixsort(const u_char **base, int nmemb, u_char *table, u_int endbyte);
```

DESCRIPTION

The **radixsort**() and **sradixsort**() functions are implementations of radix sort.

These functions sort an *nmemb* element array of pointers to byte strings, with the initial member of which is referenced by *base*. The byte strings may contain any values. End of strings is denoted by character which has same weight as user specified value *endbyte*. *endbyte* has to be between 0 and 255.

Applications may specify a sort order by providing the *table* argument. If non-NULL, *table* must reference an array of UCHAR_MAX + 1 bytes which contains the sort weight of each possible byte value. The end-of-string byte must have a sort weight of 0 or 255 (for sorting in reverse order). More than one byte may have the same sort weight. The *table* argument is useful for applications which wish to sort different characters equally, for example, providing a table with the same weights for A-Z as for a-z will result in a case-insensitive sort. If *table* is NULL, the contents of the array are sorted in ascending order according to the ASCII order of the byte strings they reference and *endbyte* has a sorting weight of 0.

The **sradixsort**() function is stable, that is, if two elements compare as equal, their order in the sorted array is unchanged. The **sradixsort**() function uses additional memory sufficient to hold *nmemb* pointers.

The **radixsort**() function is not stable, but uses no additional memory.

These functions are variants of most-significant-byte radix sorting; in particular, see D.E. Knuth's Algorithm R and section 5.2.5, exercise 10. They take linear time relative to the number of bytes in the strings.

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] The value of the *endbyte* element of *table* is not 0 or 255.

Additionally, the **sradixsort**() function may fail and set *errno* for any of the errors specified for the library routine malloc(3).

SEE ALSO

sort(1), qsort(3)

Knuth, D.E., "Sorting and Searching", *The Art of Computer Programming*, Vol. 3, pp. 170-178, 1968.

Paige, R., "Three Partition Refinement Algorithms", *SIAM J. Comput.*, No. 6, Vol. 16, 1987.

McIlroy, P., "Computing Systems", *Engineering Radix Sort*, Vol. 6:1, pp. 5-27, 1993.

HISTORY

The **radixsort**() function first appeared in 4.4BSD.

NAME

raise — send a signal to the current process

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

int
raise(int sig);
```

DESCRIPTION

The **raise**() function sends the signal *sig* to the current process.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **raise**() function may fail and set *errno* for any of the errors specified for the library functions getpid(2) and kill(2).

SEE ALSO

kill(2)

STANDARDS

The **raise**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

raise_default_signal — raise the default signal handler

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
raise_default_signal(int sig);
```

DESCRIPTION

The **raise_default_signal**() function raises the default signal handler for the signal *sig*. This function may be used by a user-defined signal handler router to ensure that a parent process receives the correct notification of a process termination by a signal. This can be used to avoid a common programming mistake when terminating a process from a custom SIGINT or SIGQUIT signal handler.

The operations performed are:

1. Block all signals, using sigprocmask(3).
2. Set the signal handler for signal *sig* to the default signal handler (SIG_DFL).
3. raise(3) signal *sig*.
4. Unblock signal *sig* to deliver it.
5. Restore the original signal mask and handler, even if there was a failure.

See signal(7) for a table of signals and default actions.

The **raise_default_signal**() function should be async-signal-safe.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **raise_default_signal**() function may fail and set *errno* for any of the errors specified for the functions sigemptyset(3), sigfillset(3), sigaddset(3), sigprocmask(2), sigaction(2), or raise(3).

SEE ALSO

sigaction(2), sigprocmask(2), raise(3), signal(7)

HISTORY

The **raise_default_signal**() function first appeared in NetBSD 5.0.

NAME

rand, **srand**, **rand_r** — bad random number generator

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

void
srand(unsigned int seed);

int
rand(void);

int
rand_r(unsigned int *seed);
```

DESCRIPTION

These interfaces are obsoleted by random(3).

The **rand**() function computes a sequence of pseudo-random integers in the range of 0 to RAND_MAX (as defined by the header file ⟨stdlib.h⟩).

The **srand**() function sets its argument as the seed for a new sequence of pseudo-random numbers to be returned by **rand**(). These sequences are repeatable by calling **srand**() with the same seed value.

If no seed value is provided, the **rand**() function is automatically seeded with a value of 1.

The **rand_r**() function is a reentrant interface to **rand**(); the seed has to be supplied and is maintained by the caller.

SEE ALSO

random(3), rnd(4)

STANDARDS

The **rand**() and **srand**() functions conform to ANSI X3.159-1989 (“ANSI C89”). The **rand_r**() function conforms to IEEE Std 1003.1c-1995 (“POSIX.1”).

NAME

drand48, **erand48**, **lrand48**, **nrand48**, **mrand48**, **jrand48**, **srand48**, **seed48**, **lcong48** — pseudo-random number generators and initialization routines

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

double
drand48(void);

double
erand48(unsigned short xseed[3]);

long
lrand48(void);

long
nrand48(unsigned short xseed[3]);

long
mrand48(void);

long
jrand48(unsigned short xseed[3]);

void
srand48(long seed);

unsigned short *
seed48(unsigned short xseed[3]);

void
lcong48(unsigned short p[7]);
```

DESCRIPTION

The **rand48**() family of functions generates pseudo-random numbers using a linear congruential algorithm working on integers 48 bits in size. The particular formula employed is r(n+1) = (a * r(n) + c) mod m where the default values are for the multiplicand a = 0x5deece66d = 25214903917 and the addend c = 0xb = 11. The modulus is always fixed at m = 2 ** 48. r(n) is called the seed of the random number generator.

For all the six generator routines described next, the first computational step is to perform a single iteration of the algorithm.

drand48() and **erand48**() return values of type double. The full 48 bits of r(n+1) are loaded into the mantissa of the returned value, with the exponent set such that the values produced lie in the interval [0.0, 1.0).

lrand48() and **nrand48**() return values of type long in the range [0, 2**31-1]. The high-order (31) bits of r(n+1) are loaded into the lower bits of the returned value, with the topmost (sign) bit set to zero.

mrand48() and **jrand48**() return values of type long in the range [-2**31, 2**31-1]. The high-order (32) bits of r(n+1) are loaded into the returned value.

drand48(), **lrand48**(), and **mrand48**() use an internal buffer to store r(n). For these functions the initial value of r(0) = 0x1234abcd330e = 20017429951246.

On the other hand, **erand48**(), **nrand48**(), and **jrand48**() use a user-supplied buffer to store the seed r(n), which consists of an array of 3 shorts, where the zeroth member holds the least significant bits.

All functions share the same multiplicand and addend.

srand48() is used to initialize the internal buffer r(n) of **drand48**(), **lrand48**(), and **mrand48**() such that the 32 bits of the seed value are copied into the upper 32 bits of r(n), with the lower 16 bits of r(n) arbitrarily being set to 0x330e. Additionally, the constant multiplicand and addend of the algorithm are reset to the default values given above.

seed48() also initializes the internal buffer r(n) of **drand48**(), **lrand48**(), and **mrand48**(), but here all 48 bits of the seed can be specified in an array of 3 shorts, where the zeroth member specifies the lowest bits. Again, the constant multiplicand and addend of the algorithm are reset to the default values given above. **seed48**() returns a pointer to an array of 3 shorts which contains the old seed. This array is statically allocated, thus its contents are lost after each new call to **seed48**().

Finally, **lcong48**() allows full control over the multiplicand and addend used in **drand48**(), **erand48**(), **lrand48**(), **nrand48**(), **mrand48**(), and **jrand48**(), and the seed used in **drand48**(), **lrand48**(), and **mrand48**(). An array of 7 shorts is passed as parameter; the first three shorts are used to initialize the seed; the second three are used to initialize the multiplicand; and the last short is used to initialize the addend. It is thus not possible to use values greater than 0xffff as the addend.

Note that all three methods of seeding the random number generator always also set the multiplicand and addend for any of the six generator calls.

For a more powerful random number generator, see `random`(3).

SEE ALSO

`rand`(3), `random`(3)

AUTHORS

Martin Birgmeier

NAME

random, **srandom**, **initstate**, **setstate** — better random number generator; routines for changing generators

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

long
random(void);

void
srandom(unsigned long seed);

char *
initstate(unsigned long seed, char *state, size_t n);

char *
setstate(char *state);
```

DESCRIPTION

The **random**() function uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random number generator is very large, approximately $16\times(2^{31}-1)$. The maximum value RANDOM_MAX is defined in <stdlib.h>.

The **random**() and **srandom**() have (almost) the same calling sequence and initialization properties as rand(3) and srand(3). The difference is that rand(3) produces a much less random sequence — in fact, the low dozen bits generated by rand(3) go through a cyclic pattern. All the bits generated by **random**() are usable. For example, '`random()&01`' will produce a random binary value.

Like rand(3), **random**() will by default produce a sequence of numbers that can be duplicated by calling **srandom**() with '`1`' as the seed.

The **initstate**() routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by **initstate**() to decide how sophisticated a random number generator it should use — the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error). The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. The state array passed to **initstate**() must be aligned to a 32-bit boundary. This can be achieved by using a suitably-sized array of ints, and casting the array to char * when passing it to **initstate**(). The **initstate**() function returns a pointer to the previous state information array.

Once a state has been initialized, the **setstate**() routine provides for rapid switching between states. The **setstate**() function returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to **initstate**() or **setstate**().

Once a state array has been initialized, it may be restarted at a different point either by calling **initstate**() (with the desired seed, the state array, and its size) or by calling both **setstate**() (with the state array) and **srandom**() (with the desired seed). The advantage of calling both **setstate**() and **srandom**() is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69}, which should be sufficient for most purposes.

DIAGNOSTICS

If **initstate**() is called with less than 8 bytes of state information, or if **setstate**() detects that the state information has been garbled, error messages are printed on the standard error output.

SEE ALSO

rand(3), srand(3), rnd(4), rnd(9)

HISTORY

These functions appeared in 4.2BSD.

AUTHORS

Earl T. Cohen

BUGS

About 2/3 the speed of rand(3).

NAME

randomid **randomid_new**, **randomid_delete**, — provide pseudo-random data stream without repetitions

SYNOPSIS

```
#include <sys/types.h>
#include <randomid.h>

uint32_t
randomid(randomid_t ctx);

randomid_t
randomid_new(int bits, long timeo);

void
randomid_delete(randomid_t ctx);
```

DESCRIPTION

The **randomid**() function provides pseudo-random data stream, which is guaranteed not to generate the same number twice during a certain duration. *ctx* is the context which holds internal state for the random number generator.

To initialize a context, *randomid_new* is used. *bits* specifies the bitwidth of the value generated by **randomid**(). Currently 32, 20, and 16 are supported. *timeo* specifies the reinitialization interval in seconds. *timeo* has to be bigger than RANDOMID_TIMEO_MIN. *randomid_new* returns a dynamically-allocated memory region allocated by malloc(3).

randomid_delete() will free(3) the internal state *ctx*.

The same number may appear after two reinitialization events of the internal state, *ctx*. Reinitialization happens when the random number generator cycle is exhausted, or *timeo* seconds have passed since the last reinitialization. For instance, *ctx* configured to generate 16 bit data stream will reinitialize its internal state every 30000 calls to **randomid**() (or after *timeo* seconds), therefore the same data will not appear until after 30000 calls to **randomid**() (or after *timeo* seconds).

The internal state, *ctx*, determines the data stream generated by **randomid**(). *ctx* must be allocated per data stream (such as a specific data field). It must not be shared among multiple data streams with different usage.

EXAMPLES

```
#include <stdio.h>
#include <sys/types.h>
#include <randomid.h>

uint32_t
genid(void)
{
        static randomid_t ctx = NULL;

        if (!ctx)
                ctx = randomid_new(16, (long)3600);
        return randomid(ctx);
}
```

ERRORS

randomid_new() returns NULL on error and sets the external variable *errno*.

SEE ALSO

arc4random(3)

HISTORY

The pseudo-random data stream generator was designed by Niels Provos for OpenBSD IPv4 fragment ID generation. **randomid**() is a generalized version of the generator, reworked by Jun-ichiro itojun Hagino, and was introduced in NetBSD 2.0.

NAME

rcmd, **orcmd**, **rcmd_af**, **orcmd_af**, **rresvport**, **rresvport_af**, **iruserok**, **ruserok**, **iruserok_sa** — routines for returning a stream to a remote command

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
rcmd(char **ahost, int inport, const char *locuser, const char *remuser,
    const char *cmd, int *fd2p);

int
orcmd(char **ahost, int inport, const char *locuser, const char *remuser,
    const char *cmd, int *fd2p);

int
rcmd_af(char **ahost, int inport, const char *locuser, const char *remuser,
    const char *cmd, int *fd2p, int af);

int
orcmd_af(char **ahost, int inport, const char *locuser, const char *remuser,
    const char *cmd, int *fd2p, int af);

int
rresvport(int *port);

int
rresvport_af(int *port, int family);

int
iruserok(uint32_t raddr, int superuser, const char *ruser,
    const char *luser);

int
ruserok(const char *rhost, int superuser, const char *ruser,
    const char *luser);

int
iruserok_sa(const void *raddr, int rlen, int superuser, const char *ruser,
    const char *luser);
```

DESCRIPTION

The **rcmd**() function is available for use by anyone to run commands on a remote system. It acts like the **orcmd**() command, with the exception that it makes a call out to the rcmd(1) command, or any other user-specified command, to perform the actual connection (thus not requiring that the caller be running as the super-user), and is only available for the “shell/tcp” port. The **orcmd**() function is used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. While **rcmd**() and **orcmd**() can only handle IPv4 address in the first argument, **rcmd_af**() and **orcmd_af**() can handle other cases as well. The **rresvport**() function returns a descriptor to a socket with an address in the privileged port space. The **rresvport_af**() function is similar to **rresvport**(), but you can explicitly specify the address family to use. Calling **rresvport_af**() with AF_INET has the same effect as **rresvport**(). The **iruserok**() and **ruserok**() functions are used by servers to authenticate clients requesting service with **rcmd**(). All six functions are present in the same file and are used by the

rshd(8) server (among others). **iruserok_sa**() is an address family independent variant of **iruserok**().

The **rcmd**() function looks up the host **ahost* using gethostbyname(3), returning −1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as *stdin* and *stdout*. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the *stderr* (unit 2 of the remote command) will be made the same as the *stdout* and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

rcmd_af() and **orcmd_af**() take address family in the last argument. If the last argument is PF_UNSPEC, interpretation of **ahost* will obey the underlying address resolution like DNS.

The protocol is described in detail in rshd(8).

The **rresvport**() and **rresvport_af**() functions are used to obtain a socket with a privileged address bound to it. This socket is suitable for use by **rcmd**() and several other functions. Privileged Internet ports are those in the range 0 to 1023. Only the super-user is allowed to bind an address of this sort to a socket.

The **iruserok**() and **ruserok**() functions take a remote host's IP address or name, respectively, two user names and a flag indicating whether the local user's name is that of the super-user. Then, if the user is *NOT* the super-user, it checks the /etc/hosts.equiv file. If that lookup is not done, or is unsuccessful, the .rhosts in the local user's home directory is checked to see if the request for service is allowed.

If this file does not exist, is not a regular file, is owned by anyone other than the user or the super-user, or is writable by anyone other than the owner, the check automatically fails. Zero is returned if the machine name is listed in the "hosts.equiv" file, or the host and remote user name are found in the ".rhosts" file; otherwise **iruserok**() and **ruserok**() return −1. If the local domain (as obtained from gethostname(3)) is the same as the remote domain, only the machine name need be specified.

If the IP address of the remote host is known, **iruserok**() should be used in preference to **ruserok**(), as it does not require trusting the DNS server for the remote host's domain.

While **iruserok**() can handle IPv4 addresses only, **iruserok_sa**() and **ruserok**() can handle other address families as well, like IPv6. The first argument of **iruserok_sa**() is typed as *void ** to avoid dependency between ⟨unistd.h⟩ and ⟨sys/socket.h⟩.

ENVIRONMENT

RCMD_CMD When using the **rcmd**() function, this variable is used as the program to run instead of rcmd(1).

DIAGNOSTICS

The **rcmd**() function returns a valid socket descriptor on success. It returns −1 on error and prints a diagnostic message on the standard error.

The **rresvport**() and **rresvport_af**() function return a valid, bound socket descriptor on success. They return −1 on error with the global value *errno* set according to the reason for failure. The error code EAGAIN is overloaded to mean ''All network ports in use.''

SEE ALSO

`rcmd`(1), `rlogin`(1), `rsh`(1), `intro`(2), `rexec`(3), `hosts.equiv`(5), `rhosts`(5), `rexecd`(8), `rlogind`(8), `rshd`(8)

HISTORY

The **`orcmd`**(), **`rresvport`**(), **`iruserok`**() and **`ruserok`**() functions appeared in 4.2BSD, where the **`orcmd`**() function was called **`rcmd`**(). The (newer) **`rcmd`**() function appeared in NetBSD 1.3. **`rcmd_af`**() and **`rresvport_af`**() were defined in RFC2292.

NAME

re_comp, **re_exec** — regular expression handler

LIBRARY

Compatibility Library (libcompat, −lcompat)

SYNOPSIS

```
#include <re_comp.h>

char *
re_comp(const char *s);

int
re_exec(const char *s);
```

DESCRIPTION

This interface is made obsolete by regex(3). It is available from the compatibility library, libcompat.

The **re_comp**() function compiles a string into an internal form suitable for pattern matching. The **re_exec**() function checks the argument string against the last string passed to **re_comp**().

The **re_comp**() function returns 0 if the string *s* was compiled successfully; otherwise a string containing an error message is returned. If **re_comp**() is passed 0 or a null string, it returns without changing the currently compiled regular expression.

The **re_exec**() function returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and −1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both **re_comp**() and **re_exec**() may have trailing or embedded newline characters; they are terminated by NULs. The regular expressions recognized are described in the manual entry for ed(1), given the above difference.

DIAGNOSTICS

The **re_exec**() function returns −1 for an internal error.

The **re_comp**() function returns one of the following strings if an error occurs:

No previous regular expression,
Regular expression too long,
unmatched \(,
missing],
too many \(\) pairs,
unmatched \).

SEE ALSO

ed(1), egrep(1), ex(1), fgrep(1), grep(1), regex(3)

HISTORY

The **re_comp**() and **re_exec**() functions appeared in 4.0BSD.

NAME

readline − get a line from a user with editing

SYNOPSIS

#include <stdio.h>
#include <readline/readline.h>
#include <readline/history.h>

*char **
readline (*const char *prompt*);

COPYRIGHT

DESCRIPTION

readline will read a line from the terminal and return it, using **prompt** as a prompt. If **prompt** is **NULL** or the empty string, no prompt is issued. The line returned is allocated with *malloc*(3); the caller must free it when finished. The line returned has the final newline removed, so only the text of the line remains.

readline offers editing capabilities while the user is entering the line. By default, the line editing commands are similar to those of emacs. A vi−style line editing interface is also available.

This manual page describes only the most basic use of **readline**. Much more functionality is available; see *The GNU Readline Library* and *The GNU History Library* for additional information.

RETURN VALUE

readline returns the text of the line read. A blank line returns the empty string. If **EOF** is encountered while reading a line, and the line is empty, **NULL** is returned. If an **EOF** is read with a non−empty line, it is treated as a newline.

NOTATION

An emacs-style notation is used to denote keystrokes. Control keys are denoted by C−*key*, e.g., C−n means Control−N. Similarly, *meta* keys are denoted by M−*key*, so M−x means Meta−X. (On keyboards without a *meta* key, M−*x* means ESC *x*, i.e., press the Escape key then the *x* key. This makes ESC the *meta prefix*. The combination M−C−*x* means ESC−Control−*x*, or press the Escape key then hold the Control key while pressing the *x* key.)

Readline commands may be given numeric *arguments*, which normally act as a repeat count. Sometimes, however, it is the sign of the argument that is significant. Passing a negative argument to a command that acts in the forward direction (e.g., **kill−line**) causes that command to act in a backward direction. Commands whose behavior with arguments deviates from this are noted.

When a command is described as *killing* text, the text deleted is saved for possible future retrieval (*yanking*). The killed text is saved in a *kill ring*. Consecutive kills cause the text to be accumulated into one unit, which can be yanked all at once. Commands which do not kill text separate the chunks of text on the kill ring.

INITIALIZATION FILE

Readline is customized by putting commands in an initialization file (the *inputrc* file). The name of this file is taken from the value of the **INPUTRC** environment variable. If that variable is unset, the default is *~/.inputrc*. When a program which uses the readline library starts up, the init file is read, and the key bindings and variables are set. There are only a few basic constructs allowed in the readline init file. Blank lines are ignored. Lines beginning with a **#** are comments. Lines beginning with a **$** indicate conditional constructs. Other lines denote key bindings and variable settings. Each program using this library may add its own commands and bindings.

For example, placing

M−Control−u: universal−argument

or

C−Meta−u: universal−argument

into the *inputrc* would make M−C−u execute the readline command *universal−argument*.

The following symbolic character names are recognized while processing key bindings: *DEL*, *ESC*, *ESCAPE*, *LFD*, *NEWLINE*, *RET*, *RETURN*, *RUBOUT*, *SPACE*, *SPC*, and *TAB*.

In addition to command names, readline allows keys to be bound to a string that is inserted when the key is pressed (a *macro*).

Key Bindings

The syntax for controlling key bindings in the *inputrc* file is simple. All that is required is the name of the command or the text of a macro and a key sequence to which it should be bound. The name may be specified in one of two ways: as a symbolic key name, possibly with *Meta−* or *Control−* prefixes, or as a key sequence.

When using the form **keyname**:*function-name* or *macro*, *keyname* is the name of a key spelled out in English. For example:

```
Control−u: universal−argument
Meta−Rubout: backward−kill−word
Control−o: "> output"
```

In the above example, *C−u* is bound to the function **universal−argument**, *M-DEL* is bound to the function **backward−kill−word**, and *C−o* is bound to run the macro expressed on the right hand side (that is, to insert the text `> output` into the line).

In the second form, **"keyseq"**:*function−name* or *macro*, **keyseq** differs from **keyname** above in that strings denoting an entire key sequence may be specified by placing the sequence within double quotes. Some GNU Emacs style key escapes can be used, as in the following example, but the symbolic character names are not recognized.

```
"\C−u": universal−argument
"\C−x\C−r": re−read−init−file
"\e[11~": "Function Key 1"
```

In this example, *C-u* is again bound to the function **universal−argument**. *C-x C-r* is bound to the function **re−read−init−file**, and *ESC [1 1 ~* is bound to insert the text `Function Key 1`.

The full set of GNU Emacs style escape sequences available when specifying key sequences is

\C− control prefix
\M− meta prefix
\e an escape character
\\ backslash
\" literal ", a double quote
\' literal ', a single quote

In addition to the GNU Emacs style escape sequences, a second set of backslash escapes is available:

\a alert (bell)
\b backspace
\d delete
\f form feed
\n newline
\r carriage return
\t horizontal tab
\v vertical tab
nnn the eight-bit character whose value is the octal value *nnn* (one to three digits)
\x*HH* the eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits)

When entering the text of a macro, single or double quotes should be used to indicate a macro definition. Unquoted text is assumed to be a function name. In the macro body, the backslash escapes described above are expanded. Backslash will quote any other character in the macro text, including " and '.

Bash allows the current readline key bindings to be displayed or modified with the **bind** builtin command. The editing mode may be switched during interactive use by using the **−o** option to the **set** builtin command. Other programs using this library provide similar mechanisms. The *inputrc* file may be edited and re-read if a program does not provide any other means to incorporate new bindings.

Variables

Readline has variables that can be used to further customize its behavior. A variable may be set in the *inputrc* file with a statement of the form

set *variable−name value*

Except where noted, readline variables can take the values **On** or **Off** (without regard to case). Unrecognized variable names are ignored. When a variable value is read, empty or null values, "on" (case-insensitive), and "1" are equivalent to **On**. All other values are equivalent to **Off**. The variables and their default values are:

bell−style (audible)
Controls what happens when readline wants to ring the terminal bell. If set to **none**, readline never rings the bell. If set to **visible**, readline uses a visible bell if one is available. If set to **audible**, readline attempts to ring the terminal's bell.

bind−tty−special−chars (On)
If set to **On**, readline attempts to bind the control characters treated specially by the kernel's terminal driver to their readline equivalents.

comment−begin ("#")
The string that is inserted in **vi** mode when the **insert−comment** command is executed. This command is bound to **M−#** in emacs mode and to **#** in vi command mode.

completion−ignore−case (Off)
If set to **On**, readline performs filename matching and completion in a case−insensitive fashion.

completion−query−items (100)
This determines when the user is queried about viewing the number of possible completions generated by the **possible−completions** command. It may be set to any integer value greater than or equal to zero. If the number of possible completions is greater than or equal to the value of this variable, the user is asked whether or not he wishes to view them; otherwise they are simply listed on the terminal. A negative value causes readline to never ask.

convert−meta (On)
If set to **On**, readline will convert characters with the eighth bit set to an ASCII key sequence by stripping the eighth bit and prefixing it with an escape character (in effect, using escape as the *meta prefix*).

disable−completion (Off)
If set to **On**, readline will inhibit word completion. Completion characters will be inserted into the line as if they had been mapped to **self-insert**.

editing−mode (emacs)
Controls whether readline begins with a set of key bindings similar to emacs or vi. **editing−mode** can be set to either **emacs** or **vi**.

enable−keypad (Off)
When set to **On**, readline will try to enable the application keypad when it is called. Some systems need this to enable the arrow keys.

expand−tilde (Off)
If set to **on**, tilde expansion is performed when readline attempts word completion.

history−preserve−point (Off)
If set to **on**, the history code attempts to place point at the same location on each history line retrieved with **previous-history** or **next-history**.

horizontal−scroll−mode (Off)
When set to **On**, makes readline use a single line for display, scrolling the input horizontally on a single screen line when it becomes longer than the screen width rather than wrapping to a new line.

input−meta (Off)
If set to **On**, readline will enable eight-bit input (that is, it will not clear the eighth bit in the characters it reads), regardless of what the terminal claims it can support. The name **meta−flag** is a synonym for this variable.

isearch−terminators (“C−[C−J”)
The string of characters that should terminate an incremental search without subsequently executing the character as a command. If this variable has not been given a value, the characters *ESC* and *C−J* will terminate an incremental search.

keymap (emacs)
Set the current readline keymap. The set of legal keymap names is *emacs*, *emacs-standard*, *emacs-meta*, *emacs-ctlx*, *vi*, *vi-move*, *vi-command*, and *vi-insert*. *vi* is equivalent to *vi-command*; *emacs* is equivalent to *emacs-standard*. The default value is *emacs*. The value of **editing−mode** also affects the default keymap.

mark−directories (On)
If set to **On**, completed directory names have a slash appended.

mark−modified−lines (Off)
If set to **On**, history lines that have been modified are displayed with a preceding asterisk (*****).

mark−symlinked−directories (Off)
If set to **On**, completed names which are symbolic links to directories have a slash appended (subject to the value of **mark−directories**).

match−hidden−files (On)
This variable, when set to **On**, causes readline to match files whose names begin with a ‘.’ (hidden files) when performing filename completion, unless the leading ‘.’ is supplied by the user in the filename to be completed.

output−meta (Off)
If set to **On**, readline will display characters with the eighth bit set directly rather than as a meta-prefixed escape sequence.

page−completions (On)
If set to **On**, readline uses an internal *more*-like pager to display a screenful of possible completions at a time.

print−completions−horizontally (Off)
If set to **On**, readline will display completions with matches sorted horizontally in alphabetical order, rather than down the screen.

show−all−if−ambiguous (Off)
This alters the default behavior of the completion functions. If set to **on**, words which have more than one possible completion cause the matches to be listed immediately instead of ringing the bell.

show−all−if−unmodified (Off)
This alters the default behavior of the completion functions in a fashion similar to **show−all−if−ambiguous**. If set to **on**, words which have more than one possible completion without any possible partial completion (the possible completions don’t share a common prefix) cause the matches to be listed immediately instead of ringing the bell.

visible−stats (Off)
If set to **On**, a character denoting a file’s type as reported by *stat*(2) is appended to the filename when listing possible completions.

Conditional Constructs

Readline implements a facility similar in spirit to the conditional compilation features of the C preprocessor which allows key bindings and variable settings to be performed as the result of tests. There are four parser directives used.

$if The **$if** construct allows bindings to be made based on the editing mode, the terminal being used, or the application using readline. The text of the test extends to the end of the line; no characters are required to isolate it.

mode The **mode=** form of the **$if** directive is used to test whether readline is in emacs or vi mode. This may be used in conjunction with the **set keymap** command, for instance, to set bindings in the *emacs-standard* and *emacs-ctlx* keymaps only if readline is starting out in emacs mode.

term The **term=** form may be used to include terminal-specific key bindings, perhaps to bind the key sequences output by the terminal's function keys. The word on the right side of the **=** is tested against the full name of the terminal and the portion of the terminal name before the first **-**. This allows *sun* to match both *sun* and *sun-cmd*, for instance.

application
The **application** construct is used to include application-specific settings. Each program using the readline library sets the *application name*, and an initialization file can test for a particular value. This could be used to bind key sequences to functions useful for a specific program. For instance, the following command adds a key sequence that quotes the current or previous word in Bash:

$if Bash
Quote the current or previous word
"\C-xq": "\eb\"\ef\""
$endif

$endif This command, as seen in the previous example, terminates an **$if** command.

$else Commands in this branch of the **$if** directive are executed if the test fails.

$include
This directive takes a single filename as an argument and reads commands and bindings from that file. For example, the following directive would read */etc/inputrc*:

$include */etc/inputrc*

SEARCHING

Readline provides commands for searching through the command history for lines containing a specified string. There are two search modes: *incremental* and *non-incremental*.

Incremental searches begin before the user has finished typing the search string. As each character of the search string is typed, readline displays the next entry from the history matching the string typed so far. An incremental search requires only as many characters as needed to find the desired history entry. To search backward in the history for a particular string, type **C-r**. Typing **C-s** searches forward through the history. The characters present in the value of the **isearch-terminators** variable are used to terminate an incremental search. If that variable has not been assigned a value the *Escape* and **C-J** characters will terminate an incremental search. **C-G** will abort an incremental search and restore the original line. When the search is terminated, the history entry containing the search string becomes the current line.

To find other matching entries in the history list, type **C-s** or **C-r** as appropriate. This will search backward or forward in the history for the next line matching the search string typed so far. Any other key sequence bound to a readline command will terminate the search and execute that command. For instance, a newline will terminate the search and accept the line, thereby executing the command from the history list. A movement command will terminate the search, make the last line found the current line, and begin editing.

Non-incremental searches read the entire search string before starting to search for matching history lines. The search string may be typed by the user or be part of the contents of the current line.

EDITING COMMANDS

The following is a list of the names of the commands and the default key sequences to which they are bound. Command names without an accompanying key sequence are unbound by default.

In the following descriptions, *point* refers to the current cursor position, and *mark* refers to a cursor position saved by the **set-mark** command. The text between the point and mark is referred to as the *region*.

Commands for Moving

beginning−of−line (C−a)
Move to the start of the current line.

end−of−line (C−e)
Move to the end of the line.

forward−char (C−f)
Move forward a character.

backward−char (C−b)
Move back a character.

forward−word (M−f)
Move forward to the end of the next word. Words are composed of alphanumeric characters (letters and digits).

backward−word (M−b)
Move back to the start of the current or previous word. Words are composed of alphanumeric characters (letters and digits).

clear−screen (C−l)
Clear the screen leaving the current line at the top of the screen. With an argument, refresh the current line without clearing the screen.

redraw−current−line
Refresh the current line.

Commands for Manipulating the History

accept−line (Newline, Return)
Accept the line regardless of where the cursor is. If this line is non-empty, it may be added to the history list for future recall with **add_history()**. If the line is a modified history line, the history line is restored to its original state.

previous−history (C−p)
Fetch the previous command from the history list, moving back in the list.

next−history (C−n)
Fetch the next command from the history list, moving forward in the list.

beginning−of−history (M−<)
Move to the first line in the history.

end−of−history (M−>)
Move to the end of the input history, i.e., the line currently being entered.

reverse−search−history (C−r)
Search backward starting at the current line and moving ‘up’ through the history as necessary. This is an incremental search.

forward−search−history (C−s)
Search forward starting at the current line and moving ‘down’ through the history as necessary. This is an incremental search.

non−incremental−reverse−search−history (M−p)
Search backward through the history starting at the current line using a non-incremental search for a string supplied by the user.

non−incremental−forward−search−history (M−n)
Search forward through the history using a non-incremental search for a string supplied by the user.

history−search−forward
Search forward through the history for the string of characters between the start of the current line and the current cursor position (the *point*). This is a non-incremental search.

history−search−backward
Search backward through the history for the string of characters between the start of the current line and the point. This is a non-incremental search.

yank−nth−arg (M−C−y)
Insert the first argument to the previous command (usually the second word on the previous line) at point. With an argument *n*, insert the *n*th word from the previous command (the words in the

previous command begin with word 0). A negative argument inserts the *n*th word from the end of the previous command. Once the argument *n* is computed, the argument is extracted as if the "!*n*" history expansion had been specified.

yank−last−arg (M−., M−_)

Insert the last argument to the previous command (the last word of the previous history entry). With an argument, behave exactly like **yank−nth−arg**. Successive calls to **yank−last−arg** move back through the history list, inserting the last argument of each line in turn. The history expansion facilities are used to extract the last argument, as if the "!$" history expansion had been specified.

Commands for Changing Text

delete−char (C−d)

Delete the character at point. If point is at the beginning of the line, there are no characters in the line, and the last character typed was not bound to **delete−char**, then return **EOF**.

backward−delete−char (Rubout)

Delete the character behind the cursor. When given a numeric argument, save the deleted text on the kill ring.

forward−backward−delete−char

Delete the character under the cursor, unless the cursor is at the end of the line, in which case the character behind the cursor is deleted.

quoted−insert (C−q, C−v)

Add the next character that you type to the line verbatim. This is how to insert characters like **C−q**, for example.

tab−insert (M-TAB)

Insert a tab character.

self−insert (a, b, A, 1, !, ...)

Insert the character typed.

transpose−chars (C−t)

Drag the character before point forward over the character at point, moving point forward as well. If point is at the end of the line, then this transposes the two characters before point. Negative arguments have no effect.

transpose−words (M−t)

Drag the word before point past the word after point, moving point over that word as well. If point is at the end of the line, this transposes the last two words on the line.

upcase−word (M−u)

Uppercase the current (or following) word. With a negative argument, uppercase the previous word, but do not move point.

downcase−word (M−l)

Lowercase the current (or following) word. With a negative argument, lowercase the previous word, but do not move point.

capitalize−word (M−c)

Capitalize the current (or following) word. With a negative argument, capitalize the previous word, but do not move point.

overwrite−mode

Toggle overwrite mode. With an explicit positive numeric argument, switches to overwrite mode. With an explicit non-positive numeric argument, switches to insert mode. This command affects only **emacs** mode; **vi** mode does overwrite differently. Each call to *readline()* starts in insert mode. In overwrite mode, characters bound to **self−insert** replace the text at point rather than pushing the text to the right. Characters bound to **backward−delete−char** replace the character before point with a space. By default, this command is unbound.

Killing and Yanking

kill−line (C−k)

Kill the text from point to the end of the line.

backward−kill−line (C−x Rubout)
Kill backward to the beginning of the line.

unix−line−discard (C−u)
Kill backward from point to the beginning of the line. The killed text is saved on the kill-ring.

kill−whole−line
Kill all characters on the current line, no matter where point is.

kill−word (M−d)
Kill from point the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as those used by **forward−word**.

backward−kill−word (M−Rubout)
Kill the word behind point. Word boundaries are the same as those used by **backward−word**.

unix−word−rubout (C−w)
Kill the word behind point, using white space as a word boundary. The killed text is saved on the kill-ring.

unix−filename−rubout
Kill the word behind point, using white space and the slash character as the word boundaries. The killed text is saved on the kill-ring.

delete−horizontal−space (M−\)
Delete all spaces and tabs around point.

kill−region
Kill the text between the point and *mark* (saved cursor position). This text is referred to as the *region*.

copy−region−as−kill
Copy the text in the region to the kill buffer.

copy−backward−word
Copy the word before point to the kill buffer. The word boundaries are the same as **backward−word**.

copy−forward−word
Copy the word following point to the kill buffer. The word boundaries are the same as **forward−word**.

yank (C−y)
Yank the top of the kill ring into the buffer at point.

yank−pop (M−y)
Rotate the kill ring, and yank the new top. Only works following **yank** or **yank−pop**.

Numeric Arguments

digit−argument (M−0, M−1, ..., M−−)
Add this digit to the argument already accumulating, or start a new argument. M−− starts a negative argument.

universal−argument
This is another way to specify an argument. If this command is followed by one or more digits, optionally with a leading minus sign, those digits define the argument. If the command is followed by digits, executing **universal−argument** again ends the numeric argument, but is otherwise ignored. As a special case, if this command is immediately followed by a character that is neither a digit or minus sign, the argument count for the next command is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four, a second time makes the argument count sixteen, and so on.

Completing

complete (TAB)
Attempt to perform completion on the text before point. The actual completion performed is application-specific. **Bash**, for instance, attempts completion treating the text as a variable (if the text begins with **$**), username (if the text begins with **~**), hostname (if the text begins with **@**), or command (including aliases and functions) in turn. If none of these produces a match, filename completion is attempted. **Gdb**, on the other hand, allows completion of program functions and variables, and only attempts filename completion under certain circumstances.

possible−completions (M−?)
List the possible completions of the text before point.

insert−completions (M−*)
Insert all completions of the text before point that would have been generated by **possible−completions**.

menu−complete
Similar to **complete**, but replaces the word to be completed with a single match from the list of possible completions. Repeated execution of **menu−complete** steps through the list of possible completions, inserting each match in turn. At the end of the list of completions, the bell is rung (subject to the setting of **bell−style**) and the original text is restored. An argument of *n* moves *n* positions forward in the list of matches; a negative argument may be used to move backward through the list. This command is intended to be bound to **TAB**, but is unbound by default.

delete−char−or−list
Deletes the character under the cursor if not at the beginning or end of the line (like **delete-char**). If at the end of the line, behaves identically to **possible-completions**.

Keyboard Macros

start−kbd−macro (C−x (
Begin saving the characters typed into the current keyboard macro.

end−kbd−macro (C−x))
Stop saving the characters typed into the current keyboard macro and store the definition.

call−last−kbd−macro (C−x e)
Re-execute the last keyboard macro defined, by making the characters in the macro appear as if typed at the keyboard.

Miscellaneous

re−read−init−file (C−x C−r)
Read in the contents of the *inputrc* file, and incorporate any bindings or variable assignments found there.

abort (C−g)
Abort the current editing command and ring the terminal's bell (subject to the setting of **bell−style**).

do−uppercase−version (M−a, M−b, M−*x*, ...)
If the metafied character *x* is lowercase, run the command that is bound to the corresponding uppercase character.

prefix−meta (ESC)
Metafy the next character typed. **ESC f** is equivalent to **Meta−f**.

undo (C−_, C−x C−u)
Incremental undo, separately remembered for each line.

revert−line (M−r)
Undo all changes made to this line. This is like executing the **undo** command enough times to return the line to its initial state.

tilde−expand (M−&)
Perform tilde expansion on the current word.

set−mark (C−@, M−<space>)
Set the mark to the point. If a numeric argument is supplied, the mark is set to that position.

exchange−point−and−mark (C−x C−x)
Swap the point with the mark. The current cursor position is set to the saved position, and the old cursor position is saved as the mark.

character−search (C−])
A character is read and point is moved to the next occurrence of that character. A negative count searches for previous occurrences.

character−search−backward (M−C−])
A character is read and point is moved to the previous occurrence of that character. A negative count searches for subsequent occurrences.

insert−comment (M−#)
Without a numeric argument, the value of the readline **comment−begin** variable is inserted at the beginning of the current line. If a numeric argument is supplied, this command acts as a toggle: if the characters at the beginning of the line do not match the value of **comment−begin**, the value is inserted, otherwise the characters in **comment-begin** are deleted from the beginning of the line. In either case, the line is accepted as if a newline had been typed. The default value of **comment−begin** makes the current line a shell comment. If a numeric argument causes the comment character to be removed, the line will be executed by the shell.

dump−functions
Print all of the functions and their key bindings to the readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file.

dump−variables
Print all of the settable variables and their values to the readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file.

dump−macros
Print all of the readline key sequences bound to macros and the strings they output. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file.

emacs−editing−mode (C−e)
When in **vi** command mode, this causes a switch to **emacs** editing mode.

vi−editing−mode (M−C−j)
When in **emacs** editing mode, this causes a switch to **vi** editing mode.

DEFAULT KEY BINDINGS

The following is a list of the default emacs and vi bindings. Characters with the eighth bit set are written as M−<character>, and are referred to as *metafied* characters. The printable ASCII characters not mentioned in the list of emacs standard bindings are bound to the **self−insert** function, which just inserts the given character into the input line. In vi insertion mode, all characters not specifically mentioned are bound to **self−insert**. Characters assigned to signal generation by *stty*(1) or the terminal driver, such as C-Z or C-C, retain that function. Upper and lower case metafied characters are bound to the same function in the emacs mode meta keymap. The remaining characters are unbound, which causes readline to ring the bell (subject to the setting of the **bell−style** variable).

Emacs Mode

Emacs Standard bindings

"C-@" set-mark
"C-A" beginning-of-line
"C-B" backward-char
"C-D" delete-char
"C-E" end-of-line
"C-F" forward-char
"C-G" abort
"C-H" backward-delete-char
"C-I" complete
"C-J" accept-line
"C-K" kill-line
"C-L" clear-screen
"C-M" accept-line
"C-N" next-history
"C-P" previous-history
"C-Q" quoted-insert
"C-R" reverse-search-history
"C-S" forward-search-history
"C-T" transpose-chars
"C-U" unix-line-discard

"C-V" quoted-insert
"C-W" unix-word-rubout
"C-Y" yank
"C-]" character-search
"C-_" undo
" " to "/" self-insert
"0" to "9" self-insert
":" to "~" self-insert
"C-?" backward-delete-char

Emacs Meta bindings

"M-C-G" abort
"M-C-H" backward-kill-word
"M-C-I" tab-insert
"M-C-J" vi-editing-mode
"M-C-M" vi-editing-mode
"M-C-R" revert-line
"M-C-Y" yank-nth-arg
"M-C-[" complete
"M-C-]" character-search-backward
"M-space" set-mark
"M-#" insert-comment
"M-&" tilde-expand
"M-*" insert-completions
"M--" digit-argument
"M-." yank-last-arg
"M-0" digit-argument
"M-1" digit-argument
"M-2" digit-argument
"M-3" digit-argument
"M-4" digit-argument
"M-5" digit-argument
"M-6" digit-argument
"M-7" digit-argument
"M-8" digit-argument
"M-9" digit-argument
"M-<" beginning-of-history
"M-=" possible-completions
"M->" end-of-history
"M-?" possible-completions
"M-B" backward-word
"M-C" capitalize-word
"M-D" kill-word
"M-F" forward-word
"M-L" downcase-word
"M-N" non-incremental-forward-search-history
"M-P" non-incremental-reverse-search-history
"M-R" revert-line
"M-T" transpose-words
"M-U" upcase-word
"M-Y" yank-pop
"M-\" delete-horizontal-space
"M-~" tilde-expand
"M-C-?" backward-kill-word

"M-_" yank-last-arg

Emacs Control-X bindings

"C-XC-G" abort
"C-XC-R" re-read-init-file
"C-XC-U" undo
"C-XC-X" exchange-point-and-mark
"C-X(" start-kbd-macro
"C-X)" end-kbd-macro
"C-XE" call-last-kbd-macro
"C-XC-?" backward-kill-line

VI Mode bindings

VI Insert Mode functions

"C-D" vi-eof-maybe
"C-H" backward-delete-char
"C-I" complete
"C-J" accept-line
"C-M" accept-line
"C-R" reverse-search-history
"C-S" forward-search-history
"C-T" transpose-chars
"C-U" unix-line-discard
"C-V" quoted-insert
"C-W" unix-word-rubout
"C-Y" yank
"C-[" vi-movement-mode
"C-_" undo
" " to "~" self-insert
"C-?" backward-delete-char

VI Command Mode functions

"C-D" vi-eof-maybe
"C-E" emacs-editing-mode
"C-G" abort
"C-H" backward-char
"C-J" accept-line
"C-K" kill-line
"C-L" clear-screen
"C-M" accept-line
"C-N" next-history
"C-P" previous-history
"C-Q" quoted-insert
"C-R" reverse-search-history
"C-S" forward-search-history
"C-T" transpose-chars
"C-U" unix-line-discard
"C-V" quoted-insert
"C-W" unix-word-rubout
"C-Y" yank
"C-_" vi-undo
" " forward-char

"#" insert-comment
"$" end-of-line
"%" vi-match
"&" vi-tilde-expand
"*" vi-complete
"+" next-history
"," vi-char-search
"-" previous-history
"." vi-redo
"/" vi-search
"0" beginning-of-line
"1" to "9" vi-arg-digit
";" vi-char-search
"=" vi-complete
"?" vi-search
"A" vi-append-eol
"B" vi-prev-word
"C" vi-change-to
"D" vi-delete-to
"E" vi-end-word
"F" vi-char-search
"G" vi-fetch-history
"I" vi-insert-beg
"N" vi-search-again
"P" vi-put
"R" vi-replace
"S" vi-subst
"T" vi-char-search
"U" revert-line
"W" vi-next-word
"X" backward-delete-char
"Y" vi-yank-to
"\" vi-complete
"^" vi-first-print
"_" vi-yank-arg
"`" vi-goto-mark
"a" vi-append-mode
"b" vi-prev-word
"c" vi-change-to
"d" vi-delete-to
"e" vi-end-word
"f" vi-char-search
"h" backward-char
"i" vi-insertion-mode
"j" next-history
"k" prev-history
"l" forward-char
"m" vi-set-mark
"n" vi-search-again
"p" vi-put
"r" vi-change-char
"s" vi-subst
"t" vi-char-search
"u" vi-undo

"w" vi-next-word
"x" vi-delete
"y" vi-yank-to
"|" vi-column
"~" vi-change-case

SEE ALSO

The Gnu Readline Library, Brian Fox and Chet Ramey
The Gnu History Library, Brian Fox and Chet Ramey
bash(1)

FILES

~/.inputrc
Individual **readline** initialization file

AUTHORS

Brian Fox, Free Software Foundation
bfox@gnu.org

Chet Ramey, Case Western Reserve University
chet@ins.CWRU.Edu

BUG REPORTS

If you find a bug in **readline,** you should report it. But first, you should make sure that it really is a bug, and that it appears in the latest version of the **readline** library that you have.

Once you have determined that a bug actually exists, mail a bug report to *bug−readline@gnu.org*. If you have a fix, you are welcome to mail that as well! Suggestions and ‘philosophical’ bug reports may be mailed to *bug-readline@gnu.org* or posted to the Usenet newsgroup **gnu.bash.bug**.

Comments and bug reports concerning this manual page should be directed to *chet@ins.CWRU.Edu*.

BUGS

It’s too big and too slow.

NAME

readpassphrase — get a passphrase from the user

SYNOPSIS

```
#include <readpassphrase.h>

char *
readpassphrase(const char *prompt, char *buf, size_t bufsiz, int flags);
```

DESCRIPTION

The **readpassphrase**() function displays a prompt to, and reads in a passphrase from, `/dev/tty`. If this file is inaccessible and the `RPP_REQUIRE_TTY` flag is not set, **readpassphrase**() displays the prompt on the standard error output and reads from the standard input. In this case it is generally not possible to turn off echo.

Up to *bufsiz* - 1 characters (one is for the NUL) are read into the provided buffer *buf*. Any additional characters and the terminating newline (or return) character are discarded.

readpassphrase() takes the following optional *flags*:

```
RPP_ECHO_OFF          turn off echo (default behavior)
RPP_ECHO_ON           leave echo on
RPP_REQUIRE_TTY              fail if there is no tty
RPP_FORCELOWER        force input to lower case
RPP_FORCEUPPER        force input to upper case
RPP_SEVENBIT          strip the high bit from input
```

The calling process should zero the passphrase as soon as possible to avoid leaving the cleartext passphrase visible in the process's address space.

RETURN VALUES

On success, **readpassphrase**() returns a pointer to the null-terminated passphrase. If the `RPP_REQUIRE_TTY` flag is set and `/dev/tty` is inaccessible, **readpassphrase**() returns a null pointer.

FILES

`/dev/tty`

EXAMPLES

The following code fragment will read a passphrase from `/dev/tty` into the buffer *passbuf*.

```
char passbuf[1024];

...

if (readpassphrase("Response: ", passbuf, sizeof(passbuf),
    RPP_REQUIRE_TTY) == NULL)
        errx(1, "unable to read passphrase");

if (compare(transform(passbuf), epass) != 0)
        errx(1, "bad passphrase");

...

memset(passbuf, 0, sizeof(passbuf));
```

SEE ALSO

`getpass`(3)

HISTORY

The **`readpassphrase`**() function first appeared in OpenBSD 2.9.

NAME

realpath — returns the canonicalized absolute pathname

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/param.h>
#include <stdlib.h>

char *
realpath(const char *pathname, char resolvedname[MAXPATHLEN]);
```

DESCRIPTION

The **realpath**() function resolves all symbolic links, extra "/" characters and references to `/./` and `/../` in *pathname*, and copies the resulting absolute pathname into the memory referenced by *resolvedname*. The *resolvedname* argument *must* refer to a buffer capable of storing at least MAXPATHLEN characters.

The **realpath**() function will resolve both absolute and relative paths and return the absolute pathname corresponding to *pathname*.

RETURN VALUES

The **realpath**() function returns *resolvedname* on success. If an error occurs, **realpath**() returns NULL, and *resolvedname* contains the pathname which caused the problem.

ERRORS

The function **realpath**() may fail and set the external variable *errno* for any of the errors specified for the library functions chdir(2), close(2), fchdir(2), lstat(2), open(2), readlink(2) and getcwd(3).

SEE ALSO

getcwd(3)

HISTORY

The **realpath**() function call first appeared in 4.4BSD.

BUGS

This implementation of **realpath**() differs slightly from the Solaris implementation. The 4.4BSD version always returns absolute pathnames, whereas the Solaris implementation will, under certain circumstances, return a relative *resolvedname* when given a relative *pathname*.

NAME

recno — record number database access method

SYNOPSIS

```
#include <sys/types.h>
#include <db.h>
```

DESCRIPTION

The routine **dbopen**() is the library interface to database files. One of the supported file formats is record number files. The general description of the database access methods is in dbopen(3), this manual page describes only the recno specific information.

The record number data structure is either variable or fixed-length records stored in a flat-file format, accessed by the logical record number. The existence of record number five implies the existence of records one through four, and the deletion of record number one causes record number five to be renumbered to record number four, as well as the cursor, if positioned after record number one, to shift down one record.

The recno access method specific data structure provided to **dbopen**() is defined in the ⟨db.h⟩ include file as follows:

```
typedef struct {
        u_long flags;
        u_int cachesize;
        u_int psize;
        int lorder;
        size_t reclen;
        u_char bval;
        char *bfname;
} RECNOINFO;
```

The elements of this structure are defined as follows:

flags The flag value is specified by or'ing any of the following values:

- R_FIXEDLEN — The records are fixed-length, not byte delimited. The structure element *reclen* specifies the length of the record, and the structure element *bval* is used as the pad character. Any records, inserted into the database, that are less than *reclen* bytes long are automatically padded.
- R_NOKEY — In the interface specified by **dbopen**(), the sequential record retrieval fills in both the caller's key and data structures. If the R_NOKEY flag is specified, the cursor routines are not required to fill in the key structure. This permits applications to retrieve records at the end of files without reading all of the intervening records.
- R_SNAPSHOT — This flag requires that a snapshot of the file be taken when **dbopen**() is called, instead of permitting any unmodified records to be read from the original file.

cachesize A suggested maximum size, in bytes, of the memory cache. This value is *only* advisory, and the access method will allocate more memory rather than fail. If *cachesize* is 0 (no size is specified) a default cache is used.

psize The recno access method stores the in-memory copies of its records in a btree. This value is the size (in bytes) of the pages used for nodes in that tree. If *psize* is 0 (no page size is specified) a page size is chosen based on the underlying file system I/O block size. See btree(3) for more information.

lorder The byte order for integers in the stored database metadata. The number should represent the order as an integer; for example, big endian order would be the number 4,321. If *lorder* is 0 (no order is specified) the current host order is used.

reclen The length of a fixed-length record.

bval The delimiting byte to be used to mark the end of a record for variable-length records, and the pad character for fixed-length records. If no value is specified, newlines ("\n") are used to mark the end of variable-length records and fixed-length records are padded with spaces.

bfname The recno access method stores the in-memory copies of its records in a btree. If bfname is non-NULL, it specifies the name of the btree file, as if specified as the file name for a **dbopen**() of a btree file.

The data part of the key/data pair used by the recno access method is the same as other access methods. The key is different. The *data* field of the key should be a pointer to a memory location of type recno_t, as defined in the ⟨db.h⟩ include file. This type is normally the largest unsigned integral type available to the implementation. The *size* field of the key should be the size of that type.

Because there can be no meta-data associated with the underlying recno access method files, any changes made to the default values (e.g., fixed record length or byte separator value) must be explicitly specified each time the file is opened.

In the interface specified by **dbopen**(), using the *put* interface to create a new record will cause the creation of multiple, empty records if the record number is more than one greater than the largest record currently in the database.

ERRORS

The **recno** access method routines may fail and set *errno* for any of the errors specified for the library routine dbopen(3) or the following:

EINVAL An attempt was made to add a record to a fixed-length database that was too large to fit.

SEE ALSO

btree(3), dbopen(3), hash(3), mpool(3)

Michael Stonebraker, Heidi Stettner, Joseph Kalash, Antonin Guttman, and Nadene Lynn, "Document Processing in a Relational Database System", *Memorandum No. UCB/ERL M82/32*, May 1982.

BUGS

Only big and little endian byte order is supported.

NAME

refuse — Re-implementation of a file system in userspace system

LIBRARY

library “librefuse”

SYNOPSIS

```
#include <fuse.h>

int
fuse_main(int argc, char **argv, const struct fuse_operations *);

int
fuse_opt_add_arg(struct fuse_args *args, const char *arg);

int
fuse_opt_parse(struct fuse_args *args, void *userdata,
    const struct fuse_opt *descriptions, fuse_opt_proc_t processingfunc);

int
fuse_teardown(struct fuse *fuse, char *mountpoint);

struct fuse *
fuse_setup(int argc, char **argv, const struct fuse_operations *ops,
    size_t opssize, char **mountpoint, int *multithreaded, int *fd);

int
puffs_fuse_node_getattr(const char *path, struct stat *attrs);

int
puffs_fuse_node_readlink(const char *path, char *output, size_t outlen);

int
puffs_fuse_node_mknod(const char *path, mode_t permissions,
    dev_t devicenumber);

int
puffs_fuse_node_mkdir(const char *path, mode_t permissions);

int
puffs_fuse_unlink(const char *path);

int
puffs_fuse_node_rmdir(const char *path);

int
puffs_fuse_node_symlink(const char *path, const char *target);

int
puffs_fuse_node_rename(const char *from, const char *to);

int
puffs_fuse_node_link(const char *from, const char *to);

int
puffs_fuse_node_chmod(const char *path, mode_t permissions);

int
puffs_fuse_node_own(const char *path, uid_t owner, gid_t group);
```

int
puffs_fuse_node_truncate(*const char *path*, *off_t newsize*);

int
puffs_fuse_node_utime(*const char *path*, *struct utimbuf *newtimes*);

int
puffs_fuse_node_open(*const char *path*, *struct fuse_file_info *fileinfo*);

int
puffs_fuse_node_read(*const char *path*, *char *buffer*, *size_t bufferlen*,
off_t startoffset, *struct fuse_file_info *fileinfo*);

int
puffs_fuse_node_write(*const char *path*, *char *buffer*, *size_t bufferlen*,
off_t startoffset, *struct fuse_file_info *fileinfo*);

int
puffs_fuse_fs_statfs(*const char *path*, *struct statvfs *vfsinfo*);

int
puffs_fuse_node_flush(*const char *path*, *struct fuse_file_info *fileinfo*);

int
puffs_fuse_node_fsync(*const char *path*, *int flags*,
*struct fuse_file_info *fileinfo*);

int
puffs_fuse_node_setxattr(*const char *path*, *const char *attrname*,
*const char *attrvalue*, *size_t attrsize*, *int flags*);

int
puffs_fuse_node_getxattr(*const char *path*, *const char *attrname*,
*const char *attrvalue*, *size_t attrvaluesize*);

int
puffs_fuse_node_listxattr(*const char *path*, *const char *attrname*,
size_t attrvaluesize);

int
puffs_fuse_node_removexattr(*const char *path*, *const char *attrname*);

int
puffs_fuse_node_opendir(*const char *path*, *struct fuse_file_info *fileinfo*);

int
puffs_fuse_node_readdir(*const char *path*, *void *data*,
fuse_fill_dir_t fillinfo, *off_t offset*,
*struct fuse_file_info *fileinfo*);

int
puffs_fuse_node_releasedir(*const char *path*,
*struct fuse_file_info *fileinfo*);

int
puffs_fuse_node_fsyncdir(*const char *path*, *int flags*,
*struct fuse_file_info *fileinfo*);

void *
puffs_fuse_fs_init(*struct fuse_conn_info *connectioninfo*);

void
puffs_fuse_node_destroy(*void *connection*);

int
puffs_fuse_node_access(*const char *path*, *int accesstype*);

int
puffs_fuse_node_create(*const char *path*, *mode_t permissions*,
*struct fuse_file_info *fileinfo*);

int
puffs_fuse_node_ftruncate(*const char *path*, *off_t offset*,
*struct fuse_file_info *fileinfo*);

int
puffs_fuse_node_fgetattr(*const char *path*, *struct stat *attrs*,
*struct fuse_file_info *fileinfo*);

int
puffs_fuse_node_lock(*const char *path*, *struct fuse_file_info *fileinfo*,
int locktype, *struct flock *lockinfo*);

int
puffs_fuse_node_utimens(*const char *path*, *const struct timespec *newtimes*);

int
puffs_fuse_node_bmap(*const char *path*, *size_t mapsize*, *uint64_t offset*);

DESCRIPTION

refuse is a reimplementation of the file system in user space subsystem. Operations are transported from the kernel virtual file system layer to the concrete implementation behind **refuse**, where they are processed and results are sent back to the kernel.

It uses the framework provided by the puffs(3) subsystem, and, through that, the kernel interface provided by puffs(4).

SEE ALSO

puffs(3), puffs(4)

Antti Kantee and Alistair Crooks, "ReFUSE: Userspace FUSE Reimplementation Using puffs", *EuroBSDCon 2007*, September 2007.

HISTORY

An unsupported experimental version of **refuse** first appeared in NetBSD 5.0.

AUTHORS

Alistair Crooks ⟨agc@NetBSD.org⟩,
Antti Kantee ⟨pooka@NetBSD.org⟩

BUGS

Many, legion, but well-loved.

NAME

regcomp, regexec, regerror, regfree − regular-expression library

SYNOPSIS

#include <sys/types.h>
#include <regex.h>

int regcomp(regex_t *preg, const char *pattern, int cflags);

int regexec(const regex_t *preg, const char *string, size_t nmatch, regmatch_t pmatch[], int eflags);

size_t regerror(int errcode, const regex_t *preg, char *errbuf, size_t errbuf_size);

void regfree(regex_t *preg);

DESCRIPTION

These routines implement POSIX 1003.2 regular expressions (''RE''s); see *re_format*(7). *Regcomp* compiles an RE written as a string into an internal form, *regexec* matches that internal form against a string and reports results, *regerror* transforms error codes from either into human-readable messages, and *regfree* frees any dynamically-allocated storage used by the internal form of an RE.

The header *<regex.h>* declares two structure types, *regex_t* and *regmatch_t*, the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type *regoff_t*, and a number of constants with names starting with ''REG_''.

Regcomp compiles the regular expression contained in the *pattern* string, subject to the flags in *cflags*, and places the results in the *regex_t* structure pointed to by *preg*. *Cflags* is the bitwise OR of zero or more of the following flags:

REG_EXTENDED — Compile modern (''extended'') REs, rather than the obsolete (''basic'') REs that are the default.

REG_BASIC — This is a synonym for 0, provided as a counterpart to REG_EXTENDED to improve readability.

REG_NOSPEC — Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the ''RE'' is a literal string. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. REG_EXTENDED and REG_NOSPEC may not be used in the same call to *regcomp*.

REG_ICASE — Compile for matching that ignores upper/lower case distinctions. See *re_format*(7).

REG_NOSUB — Compile for matching that need only report success or failure, not what was matched.

REG_NEWLINE — Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, '[^' bracket expressions and '.' never match newline, a '^' anchor matches the null string after any newline in the string in addition to its normal function, and the '$' anchor matches the null string before any newline in the string in addition to its normal function.

REG_PEND — The regular expression ends, not at the first NUL, but just before the character pointed to by the *re_endp* member of the structure pointed to by *preg*. The *re_endp* member is of type *const char **. This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

When successful, *regcomp* returns 0 and fills in the structure pointed to by *preg*. One member of that structure (other than *re_endp*) is publicized: *re_nsub*, of type *size_t*, contains the number of parenthesized subexpressions within the RE (except that the value of this member is undefined if the REG_NOSUB flag was used). If *regcomp* fails, it returns a non-zero error code; see DIAGNOSTICS.

Regexec matches the compiled RE pointed to by *preg* against the *string*, subject to the flags in *eflags*, and

reports results using *nmatch*, *pmatch*, and the returned value. The RE must have been compiled by a previous invocation of *regcomp*. The compiled form is not altered during execution of *regexec*, so a single compiled RE can be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by *string* is considered to be the text of an entire line, minus any terminating newline. The *eflags* argument is the bitwise OR of zero or more of the following flags:

REG_NOTBOL The first character of the string is not the beginning of a line, so the '^' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.

REG_NOTEOL The NUL terminating the string does not end a line, so the '$' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.

REG_STARTEND The string is considered to start at *string* + *pmatch*[0].*rm_so* and to have a terminating NUL located at *string* + *pmatch*[0].*rm_eo* (there need not actually be a NUL at that location), regardless of the value of *nmatch*. See below for the definition of *pmatch* and *nmatch*. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Note that a non-zero *rm_so* does not imply REG_NOTBOL; REG_STARTEND affects only the location of the string, not how it is matched.

See *re_format*(7) for a discussion of what is matched in situations where an RE or a portion thereof could match any of several substrings of *string*.

Normally, *regexec* returns 0 for success and the non-zero code REG_NOMATCH for failure. Other non-zero error codes may be returned in exceptional situations; see DIAGNOSTICS.

If REG_NOSUB was specified in the compilation of the RE, or if *nmatch* is 0, *regexec* ignores the *pmatch* argument (but see below for the case where REG_STARTEND is specified). Otherwise, *pmatch* points to an array of *nmatch* structures of type *regmatch_t*. Such a structure has at least the members *rm_so* and *rm_eo*, both of type *regoff_t* (a signed arithmetic type at least as large as an *off_t* and a *ssize_t*), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to *regexec*. An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0th member of the *pmatch* array is filled in to indicate what substring of *string* was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member *i* reports subexpression *i*, with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array—corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, *i* > *preg*−>*re_nsub*)—have both *rm_so* and *rm_eo* set to −1. If a subexpression participated in the match several times, the reported substring is the last one it matched. (Note, as an example in particular, that when the RE '(b*)+' matches 'bbb', the parenthesized subexpression matches each of the three 'b's and then an infinite number of empty strings following the last 'b', so the reported substring is one of the empties.)

If REG_STARTEND is specified, *pmatch* must point to at least one *regmatch_t* (even if *nmatch* is 0 or REG_NOSUB was specified), to hold the input offsets for REG_STARTEND. Use for output is still entirely controlled by *nmatch*; if *nmatch* is 0 or REG_NOSUB was specified, the value of *pmatch*[0] will not be changed by a successful *regexec*.

Regerror maps a non-zero *errcode* from either *regcomp* or *regexec* to a human-readable, printable message. If *preg* is non-NULL, the error code should have arisen from use of the *regex_t* pointed to by *preg*, and if the error code came from *regcomp*, it should have been the result from the most recent *regcomp* using that *regex_t*. (*Regerror* may be able to supply a more detailed message using information from the *regex_t*.) *Regerror* places the NUL-terminated message into the buffer pointed to by *errbuf*, limiting the length (including the NUL) to at most *errbuf_size* bytes. If the whole message won't fit, as much of it as will fit

before the terminating NUL is supplied. In any case, the returned value is the size of buffer needed to hold the whole message (including terminating NUL). If *errbuf_size* is 0, *errbuf* is ignored but the return value is still correct.

If the *errcode* given to *regerror* is first ORed with REG_ITOA, the ''message'' that results is the printable name of the error code, e.g. ''REG_NOMATCH'', rather than an explanation thereof. If *errcode* is REG_ATOI, then *preg* shall be non-NULL and the *re_endp* member of the structure it points to must point to the printable name of an error code; in this case, the result in *errbuf* is the decimal digits of the numeric value of the error code (0 if the name is not recognized). REG_ITOA and REG_ATOI are intended primarily as debugging facilities; they are extensions, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

Regfree frees any dynamically-allocated storage associated with the compiled RE pointed to by *preg*. The remaining *regex_t* is no longer a valid compiled RE and the effect of supplying it to *regexec* or *regerror* is undefined.

None of these functions references global variables except for tables of constants; all are safe for use from multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that 1003.2 leaves up to the implementor, either by explicitly saying ''undefined'' or by virtue of them being forbidden by the RE grammar. This implementation treats them as follows.

See *re_format*(7) for a discussion of the definition of case-independent matching.

There is no particular limit on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See BUGS for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by 1003.2 (such magic meanings occur only in obsolete [''basic''] REs) is taken as an ordinary character.

Any unmatched [is a REG_EBRACK error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

RE_DUP_MAX, the limit on repetition counts in bounded repetitions, is 255.

A repetition operator (?, *, +, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow '^' or '|'.

'|' cannot appear first or last in a (sub)expression or after another '|', i.e. an operand of '|' cannot be an empty subexpression. An empty parenthesized subexpression, '()', is legal and matches an empty (sub)string. An empty string is not a legal RE.

A '{' followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A '{' *not* followed by a digit is considered an ordinary character.

'^' and '$' beginning and ending subexpressions in obsolete (''basic'') REs are anchors, not ordinary characters.

SEE ALSO

grep(1), re_format(7)

POSIX 1003.2, sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

DIAGNOSTICS

Non-zero error codes from *regcomp* and *regexec* include the following:

REG_NOMATCH	regexec() failed to match
REG_BADPAT	invalid regular expression

REG_ECOLLATE	invalid collating element
REG_ECTYPE	invalid character class
REG_EESCAPE	\ applied to unescapable character
REG_ESUBREG	invalid backreference number
REG_EBRACK	brackets [ ] not balanced
REG_EPAREN	parentheses ( ) not balanced
REG_EBRACE	braces { } not balanced
REG_BADBR	invalid repetition count(s) in { }
REG_ERANGE	invalid character range in [ ]
REG_ESPACE	ran out of memory
REG_BADRPT	?, *, or + operand invalid
REG_EMPTY	empty (sub)expression
REG_ASSERT	‘‘can’t happen’’—you found a bug
REG_INVARG	invalid argument, e.g. negative-length string

HISTORY

Originally written by Henry Spencer at University of Toronto. Altered for inclusion in the 4.4BSD distribution.

BUGS

This is an alpha release with known defects. Please report problems.

There is one known functionality bug. The implementation of internationalization is incomplete: the locale is always assumed to be the default one of 1003.2, and only the collating elements etc. of that locale are available.

The back-reference code is subtle and doubts linger about its correctness in complex cases.

Regexec performance is poor. This will improve with later releases. *Nmatch* exceeding 0 is expensive; *nmatch* exceeding 1 is worse. *Regexec* is largely insensitive to RE complexity *except* that back references are massively expensive. RE length does matter; in particular, there is a strong speed bonus for keeping RE length under about 30 characters, with most special characters counting roughly double.

Regcomp implements bounded repetitions by macro expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like, say, ‘((((a{1,100}){1,100}){1,100}){1,100}){1,100}’ will (eventually) run almost any existing machine out of swap space.

There are suspected problems with response to obscure error conditions. Notably, certain kinds of internal overflow, produced only by truly enormous REs or by multiply nested bounded repetitions, are probably not handled well.

Due to a mistake in 1003.2, things like ‘a)b’ are legal REs because ‘)’ is a special character only in the presence of a previous unmatched ‘(’. This can’t be fixed until the spec is fixed.

The standard’s definition of back references is vague. For example, does ‘a\(\(b\)*\2\)*d’ match ‘abbbd’? Until the standard is clarified, behavior in such cases should not be relied on.

The implementation of word-boundary matching is a bit of a kludge, and bugs may lurk in combinations of word-boundary matching and anchoring.

NAME

regex, **regcomp**, **regexec**, **regerror**, **regfree** — regular-expression library

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <regex.h>

int
regcomp(regex_t * restrict preg, const char * restrict pattern,
    int cflags);

int
regexec(const regex_t * restrict preg, const char * restrict string,
    size_t nmatch, regmatch_t pmatch[], int eflags);

size_t
regerror(int errcode, const regex_t * restrict preg,
    char * restrict errbuf, size_t errbuf_size);

void
regfree(regex_t *preg);
```

DESCRIPTION

These routines implement IEEE Std 1003.2-1992 (“POSIX.2”) regular expressions (“RE”s); see re_format(7). **regcomp**() compiles an RE written as a string into an internal form, **regexec**() matches that internal form against a string and reports results, **regerror**() transforms error codes from either into human-readable messages, and **regfree**() frees any dynamically-allocated storage used by the internal form of an RE.

The header *<regex.h>* declares two structure types, *regex_t* and *regmatch_t*, the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type *regoff_t*, and a number of constants with names starting with “REG_”.

regcomp() compiles the regular expression contained in the *pattern* string, subject to the flags in *cflags*, and places the results in the *regex_t* structure pointed to by *preg*. *cflags* is the bitwise OR of zero or more of the following flags:

REG_EXTENDED — Compile modern (“extended”) REs, rather than the obsolete (“basic”) REs that are the default.

REG_BASIC — This is a synonym for 0, provided as a counterpart to REG_EXTENDED to improve readability.

REG_NOSPEC — Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the “RE” is a literal string. This is an extension, compatible with but not specified by IEEE Std 1003.2-1992 (“POSIX.2”), and should be used with caution in software intended to be portable to other systems. REG_EXTENDED and REG_NOSPEC may not be used in the same call to **regcomp**().

REG_ICASE — Compile for matching that ignores upper/lower case distinctions. See re_format(7).

REG_NOSUB — Compile for matching that need only report success or failure, not what was matched.

REG_NEWLINE Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, '[^' bracket expressions and '.' never match newline, a '^' anchor matches the null string after any newline in the string in addition to its normal function, and the '$' anchor matches the null string before any newline in the string in addition to its normal function.

REG_PEND The regular expression ends, not at the first NUL, but just before the character pointed to by the *re_endp* member of the structure pointed to by *preg*. The *re_endp* member is of type *const char **. This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with but not specified by IEEE Std 1003.2-1992 ("POSIX.2"), and should be used with caution in software intended to be portable to other systems.

When successful, **regcomp**() returns 0 and fills in the structure pointed to by *preg*. One member of that structure (other than *re_endp*) is publicized: *re_nsub*, of type *size_t*, contains the number of parenthesized subexpressions within the RE (except that the value of this member is undefined if the REG_NOSUB flag was used). If **regcomp**() fails, it returns a non-zero error code; see **DIAGNOSTICS**.

regexec() matches the compiled RE pointed to by *preg* against the *string*, subject to the flags in *eflags*, and reports results using *nmatch*, *pmatch*, and the returned value. The RE must have been compiled by a previous invocation of **regcomp**(). The compiled form is not altered during execution of **regexec**(), so a single compiled RE can be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by *string* is considered to be the text of an entire line, minus any terminating newline. The *eflags* argument is the bitwise OR of zero or more of the following flags:

REG_NOTBOL The first character of the string is not the beginning of a line, so the '^' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.

REG_NOTEOL The NUL terminating the string does not end a line, so the '$' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.

REG_STARTEND The string is considered to start at *string* + *pmatch[0].rm_so* and to have a terminating NUL located at *string* + *pmatch[0].rm_eo* (there need not actually be a NUL at that location), regardless of the value of *nmatch*. See below for the definition of *pmatch* and *nmatch*. This is an extension, compatible with but not specified by IEEE Std 1003.2-1992 ("POSIX.2"), and should be used with caution in software intended to be portable to other systems. Note that a non-zero *rm_so* does not imply REG_NOTBOL; REG_STARTEND affects only the location of the string, not how it is matched.

See re_format(7) for a discussion of what is matched in situations where an RE or a portion thereof could match any of several substrings of *string*.

Normally, **regexec**() returns 0 for success and the non-zero code REG_NOMATCH for failure. Other non-zero error codes may be returned in exceptional situations; see **DIAGNOSTICS**.

If REG_NOSUB was specified in the compilation of the RE, or if *nmatch* is 0, **regexec**() ignores the *pmatch* argument (but see below for the case where REG_STARTEND is specified). Otherwise, *pmatch* points to an array of *nmatch* structures of type *regmatch_t*. Such a structure has at least the members *rm_so* and *rm_eo*, both of type *regoff_t* (a signed arithmetic type at least as large as an *off_t* and a *ssize_t*), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to **regexec**(). An empty substring is denoted by equal offsets, both indicating the character follow-

ing the empty substring.

The 0th member of the *pmatch* array is filled in to indicate what substring of *string* was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member *i* reports subexpression *i*, with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array—corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, *i* > *preg->re_nsub*) —have both *rm_so* and *rm_eo* set to -1. If a subexpression participated in the match several times, the reported substring is the last one it matched. (Note, as an example in particular, that when the RE '(b*)+' matches 'bbb', the parenthesized subexpression matches each of the three 'b's and then an infinite number of empty strings following the last 'b', so the reported substring is one of the empties.)

If REG_STARTEND is specified, *pmatch* must point to at least one *regmatch_t* (even if *nmatch* is 0 or REG_NOSUB was specified), to hold the input offsets for REG_STARTEND. Use for output is still entirely controlled by *nmatch*; if *nmatch* is 0 or REG_NOSUB was specified, the value of *pmatch [0]* will not be changed by a successful **regexec**().

regerror() maps a non-zero *errcode* from either **regcomp**() or **regexec**() to a human-readable, printable message. If *preg* is non-NULL, the error code should have arisen from use of the *regex_t* pointed to by *preg*, and if the error code came from **regcomp**(), it should have been the result from the most recent **regcomp**() using that *regex_t*. (**regerror**() may be able to supply a more detailed message using information from the *regex_t*.) **regerror**() places the NUL-terminated message into the buffer pointed to by *errbuf*, limiting the length (including the NUL) to at most *errbuf_size* bytes. If the whole message won't fit, as much of it as will fit before the terminating NUL is supplied. In any case, the returned value is the size of buffer needed to hold the whole message (including terminating NUL). If *errbuf_size* is 0, *errbuf* is ignored but the return value is still correct.

If the *errcode* given to **regerror**() is first ORed with REG_ITOA, the ''message'' that results is the printable name of the error code, e.g. ''REG_NOMATCH'', rather than an explanation thereof. If *errcode* is REG_ATOI, then *preg* shall be non-NULL and the *re_endp* member of the structure it points to must point to the printable name of an error code; in this case, the result in *errbuf* is the decimal digits of the numeric value of the error code (0 if the name is not recognized). REG_ITOA and REG_ATOI are intended primarily as debugging facilities; they are extensions, compatible with but not specified by IEEE Std 1003.2-1992 ("POSIX.2"), and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

regfree() frees any dynamically-allocated storage associated with the compiled RE pointed to by *preg*. The remaining *regex_t* is no longer a valid compiled RE and the effect of supplying it to **regexec**() or **regerror**() is undefined.

None of these functions references global variables except for tables of constants; all are safe for use from multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that IEEE Std 1003.2-1992 ("POSIX.2") leaves up to the implementor, either by explicitly saying ''undefined'' or by virtue of them being forbidden by the RE grammar. This implementation treats them as follows.

See re_format(7) for a discussion of the definition of case-independent matching.

There is no particular limit on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See BUGS for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by IEEE Std 1003.2-1992 ("POSIX.2") (such magic meanings occur only in obsolete [''basic''] REs) is taken as an ordinary character.

Any unmatched [is a REG_EBRACK error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

RE_DUP_MAX, the limit on repetition counts in bounded repetitions, is 255.

A repetition operator (?, *, +, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow '^' or '|'.

'|' cannot appear first or last in a (sub)expression or after another '|', i.e. an operand of '|' cannot be an empty subexpression. An empty parenthesized subexpression, '()', is legal and matches an empty (sub)string. An empty string is not a legal RE.

A '{' followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A '{' *not* followed by a digit is considered an ordinary character.

'^' and '$' beginning and ending subexpressions in obsolete (''basic'') REs are anchors, not ordinary characters.

DIAGNOSTICS

Non-zero error codes from **regcomp**() and **regexec**() include the following:

REG_NOMATCH	regexec() failed to match
REG_BADPAT	invalid regular expression
REG_ECOLLATE	invalid collating element
REG_ECTYPE	invalid character class
REG_EESCAPE	\ applied to unescapable character
REG_ESUBREG	invalid backreference number
REG_EBRACK	brackets [ ] not balanced
REG_EPAREN	parentheses ( ) not balanced
REG_EBRACE	braces { } not balanced
REG_BADBR	invalid repetition count(s) in { }
REG_ERANGE	invalid character range in [ ]
REG_ESPACE	ran out of memory
REG_BADRPT	?, *, or + operand invalid
REG_EMPTY	empty (sub)expression
REG_ASSERT	''can't happen''—you found a bug
REG_INVARG	invalid argument, e.g. negative-length string

SEE ALSO

grep(1), sed(1), re_format(7)

IEEE Std 1003.2-1992 ("POSIX.2"), sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

HISTORY

Originally written by Henry Spencer. Altered for inclusion in the 4.4BSD distribution.

BUGS

There is one known functionality bug. The implementation of internationalization is incomplete: the locale is always assumed to be the default one of IEEE Std 1003.2-1992 ("POSIX.2"), and only the collating elements etc. of that locale are available.

The back-reference code is subtle and doubts linger about its correctness in complex cases.

regexec() performance is poor. This will improve with later releases. *nmatch* exceeding 0 is expensive; *nmatch* exceeding 1 is worse. *regexec* is largely insensitive to RE complexity *except* that back references are massively expensive. RE length does matter; in particular, there is a strong speed bonus for keeping RE length under about 30 characters, with most special characters counting roughly double.

regcomp() implements bounded repetitions by macro expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like, say, '((((a{1,100}){1,100}){1,100}){1,100}){1,100}' will (eventually) run almost any existing machine out of swap space.

There are suspected problems with response to obscure error conditions. Notably, certain kinds of internal overflow, produced only by truly enormous REs or by multiply nested bounded repetitions, are probably not handled well.

Due to a mistake in IEEE Std 1003.2-1992 ("POSIX.2"), things like 'a)b' are legal REs because ')' is a special character only in the presence of a previous unmatched '('. This can't be fixed until the spec is fixed.

The standard's definition of back references is vague. For example, does 'a\(\(b\)*\2\)*d' match 'abbbd'? Until the standard is clarified, behavior in such cases should not be relied on.

The implementation of word-boundary matching is a bit of a kludge, and bugs may lurk in combinations of word-boundary matching and anchoring.

NAME

regcomp, **regexec**, **regsub**, **regerror** — regular expression handlers

LIBRARY

Compatibility Library (libcompat, −lcompat)

SYNOPSIS

```
#include <regexp.h>

regexp *
regcomp(const char *exp);

int
regexec(const regexp *prog, const char *string);

void
regsub(const regexp *prog, const char *source, char *dest);

void
regerror(const char *msg);
```

DESCRIPTION

This interface is made obsolete by regex(**3**)**. It is available from the compatibility library, libcompat.**

The **regcomp**(), **regexec**(), **regsub**(), and **regerror**() functions implement egrep(1)-style regular expressions and supporting facilities.

The **regcomp**() function compiles a regular expression into a structure of type *regexp*, and returns a pointer to it. The space has been allocated using malloc(3) and may be released by free(3).

The **regexec**() function matches a NUL-terminated *string* against the compiled regular expression in *prog*. It returns 1 for success and 0 for failure, and adjusts the contents of *prog*'s *startp* and *endp* (see below) accordingly.

The members of a *regexp* structure include at least the following (not necessarily in order):

```
char *startp[NSUBEXP];
char *endp[NSUBEXP];
```

where NSUBEXP is defined (as 10) in the header file. Once a successful **regexec**() has been done using the **regexp**(), each *startp*- *endp* pair describes one substring within the *string*, with the *startp* pointing to the first character of the substring and the *endp* pointing to the first character following the substring. The 0th substring is the substring of *string* that matched the whole regular expression. The others are those substrings that matched parenthesized expressions within the regular expression, with parenthesized expressions numbered in left-to-right order of their opening parentheses.

The **regsub**() function copies *source* to *dest*, making substitutions according to the most recent **regexec**() performed using *prog*. Each instance of '&' in *source* is replaced by the substring indicated by *startp*[ ] and *endp*[ ]. Each instance of '*n*', where *n* is a digit, is replaced by the substring indicated by *startp*[*n*] and *endp*[*n*]. To get a literal '&' or '*n*' into *dest*, prefix it with '\'; to get a literal '\' preceding '&' or '*n*', prefix it with another '\'.

The **regerror**() function is called whenever an error is detected in **regcomp**(), **regexec**(), or **regsub**(). The default **regerror**() writes the string *msg*, with a suitable indicator of origin, on the standard error output and invokes exit(3). The **regerror**() function can be replaced by the user if other actions are desirable.

REGULAR EXPRESSION SYNTAX

A regular expression is zero or more *branches*, separated by '|'. It matches anything that matches one of the branches.

A branch is zero or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an *atom* possibly followed by '*', '+', or '?'. An atom followed by '*' matches a sequence of 0 or more matches of the atom. An atom followed by '+' matches a sequence of 1 or more matches of the atom. An atom followed by '?' matches a match of the atom, or the null string.

An atom is a regular expression in parentheses (matching a match for the regular expression), a *range* (see below), '.' (matching any single character), '^' (matching the null string at the beginning of the input string), '$' (matching the null string at the end of the input string), a '\' followed by a single character (matching that character), or a single character with no other significance (matching that character).

A *range* is a sequence of characters enclosed in '[]'. It normally matches any single character from the sequence. If the sequence begins with '^', it matches any single character *not* from the rest of the sequence. If two characters in the sequence are separated by '−', this is shorthand for the full list of ASCII characters between them (e.g. '[0-9]' matches any decimal digit). To include a literal ']' in the sequence, make it the first character (following a possible '^'). To include a literal '−', make it the first or last character.

AMBIGUITY

If a regular expression could match two different parts of the input string, it will match the one which begins earliest. If both begin in the same place but match different lengths, or match the same length in different ways, life gets messier, as follows.

In general, the possibilities in a list of branches are considered in left-to-right order, the possibilities for '*', '+', and '?' are considered longest-first, nested constructs are considered from the outermost in, and concatenated constructs are considered leftmost-first. The match that will be chosen is the one that uses the earliest possibility in the first choice that has to be made. If there is more than one choice, the next will be made in the same manner (earliest possibility) subject to the decision on the first choice. And so forth.

For example, '`(ab|a)b*c`' could match 'abc' in one of two ways. The first choice is between 'ab' and 'a'; since 'ab' is earlier, and does lead to a successful overall match, it is chosen. Since the 'b' is already spoken for, the 'b*' must match its last possibility—the empty string—since it must respect the earlier choice.

In the particular case where no '|'s are present and there is only one '*', '+', or '?', the net effect is that the longest possible match will be chosen. So '`ab*`', presented with 'xabbbby', will match 'abbbb'. Note that if '`ab*`', is tried against 'xabyabbbz', it will match 'ab' just after 'x', due to the begins-earliest rule. (In effect, the decision on where to start the match is the first choice to be made, hence subsequent choices must respect it even if this leads them to less-preferred alternatives.)

RETURN VALUES

The **`regcomp`**() function returns `NULL` for a failure (**`regerror`**() permitting), where failures are syntax errors, exceeding implementation limits, or applying '+' or '*' to a possibly-null operand.

SEE ALSO

`ed`(1), `egrep`(1), `ex`(1), `expr`(1), `fgrep`(1), `grep`(1), `regex`(3)

HISTORY

Both code and manual page for **`regcomp`**(), **`regexec`**(), **`regsub`**(), and **`regerror`**() were written at the University of Toronto and appeared in 4.3BSD−Tahoe. They are intended to be compatible with the Bell V8 `regexp`(3), but are not derived from Bell code.

BUGS

Empty branches and empty regular expressions are not portable to V8.

The restriction against applying ‘*’ or ‘+’ to a possibly-null operand is an artifact of the simplistic implementation.

Does not support `egrep`(1)’s newline-separated branches; neither does the V8 `regexp`(3), though.

Due to emphasis on compactness and simplicity, it’s not strikingly fast. It does give special attention to handling simple cases quickly.

NAME

remove — remove directory entry

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

int
remove(const char *path);
```

DESCRIPTION

The **remove**() function removes the file or directory specified by *path*.

If *path* specifies a directory, **remove**(*path*) is the equivalent of **rmdir**(*path*). Otherwise, it is the equivalent of **unlink**(*path*).

RETURN VALUES

Upon successful completion, **remove**() returns 0. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **remove**() function may fail and set *errno* for any of the errors specified for the routines rmdir(2) or unlink(2).

SEE ALSO

rmdir(2), unlink(2), symlink(7)

STANDARDS

The **remove**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

res_query, **res_search**, **res_mkquery**, **res_send**, **res_init**, **dn_comp**, **dn_expand** — resolver routines

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

res_query(const char *dname, int class, int type, u_char *answer, int anslen);

res_search(const char *dname, int class, int type, u_char *answer,
    int anslen);

res_mkquery(int op, const char *dname, int class, int type, const char *data,
    int datalen, struct rrec *newrr, char *buf, int buflen);

res_send(const u_char *msg, int msglen, u_char *answer, int anslen);

res_init();

dn_comp(const char *exp_dn, u_char *comp_dn, int length, u_char **dnptrs,
    u_char **lastdnptr);

dn_expand(const u_char *msg, const u_char *eomorig, const u_char *comp_dn,
    u_char *exp_dn, int length);
```

DESCRIPTION

These routines are used for making, sending and interpreting query and reply messages with Internet domain name servers.

Global configuration and state information that is used by the resolver routines is kept in the structure *_res*. Most of the values have reasonable defaults and can be ignored. Options stored in *_res.options* are defined in `resolv.h` and are as follows. Options are stored as a simple bit mask containing the bitwise ‘‘or’’ of the options enabled.

`RES_INIT` True if the initial name server address and default domain name are initialized (i.e., **res_init**() has been called).

`RES_DEBUG` Print debugging messages.

`RES_AAONLY` Accept authoritative answers only. With this option, **res_send**() should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

`RES_USEVC` Use TCP connections for queries instead of UDP datagrams.

`RES_STAYOPEN` Used with `RES_USEVC` to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

`RES_IGNTC` Unused currently (ignore truncation errors, i.e., don’t retry with TCP).

`RES_RECURSE` Set the recursion-desired bit in queries. This is the default. (**res_send**() does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES If set, **res_search**() will append the default domain name to single-component names (those that do not contain a dot). This option is enabled by default.

RES_DNSRCH If this option is set, **res_search**() will search for host names in the current domain and in parent domains; see hostname(7). This is used by the standard host lookup routine gethostbyname(3). This option is enabled by default.

RES_USE_INET6 Enables support for IPv6-only applications. This causes IPv4 addresses to be returned as an IPv4 mapped address. For example, 10.1.1.1 will be returned as ::ffff:10.1.1.1. The option is meaningful with certain kernel configuration only.

RES_USE_EDNS0 Enables support for OPT pseudo-RR for EDNS0 extension. With the option, resolver code will attach OPT pseudo-RR into DNS queries, to inform of our receive buffer size. The option will allow DNS servers to take advantage of non-default receive buffer size, and to send larger replies. DNS query packets with EDNS0 extension is not compatible with non-EDNS0 DNS servers.

The **res_init**() routine reads the configuration file (if any; see resolv.conf(5)) to get the default domain name, search list and the Internet address of the local name server(s). If no server is configured, the host running the resolver is tried. The current domain name is defined by the hostname if not specified in the configuration file; it can be overridden by the environment variable LOCALDOMAIN. This environment variable may contain several blank-separated tokens if you wish to override the *search list* on a per-process basis. This is similar to the *search* command in the configuration file. Another environment variable RES_OPTIONS can be set to override certain internal resolver options which are otherwise set by changing fields in the *_res* structure or are inherited from the configuration file's *options* command. The syntax of the RES_OPTIONS environment variable is explained in resolv.conf(5). Initialization normally occurs on the first call to one of the following routines.

The **res_query**() function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified *type* and *class* for the specified fully-qualified domain name *dname*. The reply message is left in the *answer* buffer with length *anslen* supplied by the caller.

The **res_search**() routine makes a query and awaits a response like **res_query**(), but in addition, it implements the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options. It returns the first successful reply.

The remaining routines are lower-level routines used by **res_query**(). The **res_mkquery**() function constructs a standard query message and places it in *buf*. It returns the size of the query, or −1 if the query is larger than *buflen*. The query type *op* is usually QUERY, but can be any of the query types defined in ⟨arpa/nameser.h⟩. The domain name for the query is given by *dname*. *newrr* is currently unused but is intended for making update messages.

The **res_send**() routine sends a pre-formatted query and returns an answer. It will call **res_init**() if RES_INIT is not set, send the query to the local name server, and handle timeouts and retries. The length of the reply message is returned, or −1 if there were errors.

The **dn_comp**() function compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or −1 if there were errors. The size of the array pointed to by *comp_dn* is given by *length*. The compression uses an array of pointers *dnptrs* to previously-compressed names in the current message. The first pointer points to the beginning of the message and the list ends with NULL. The limit to the array is specified by *lastdnptr*. A side effect of **dn_comp**() is to update the list of pointers for labels inserted into the message as the name is compressed. If *dnptr* is NULL, names are not compressed. If *lastdnptr* is NULL, the list of labels is not updated.

The **dn_expand**() entry expands the compressed domain name *comp_dn* to a full domain name. The compressed name is contained in a query or reply message; *msg* is a pointer to the beginning of the message. The uncompressed name is placed in the buffer indicated by *exp_dn* which is of size *length*. The size of compressed name is returned or −1 if there was an error.

FILES

`/etc/resolv.conf` The configuration file, see `resolv.conf`(5).

SEE ALSO

`gethostbyname`(3), `resolv.conf`(5), `hostname`(7), `named`(8)

RFC 974, *RFC 1032*, *RFC 1033*, *RFC 1034*, *RFC 1035*, *RFC 1535*

Name Server Operations Guide for BIND.

HISTORY

The **res_query** function appeared in 4.3BSD.

NAME

rexec — return stream to a remote command

LIBRARY

Compatibility Library (libcompat, −lcompat)

SYNOPSIS

```
int
rexec(char **ahost, int inport, char *user, char *passwd, char *cmd,
    int *fd2p);
```

DESCRIPTION

This interface is obsoleted by rcmd**(3). It is available from the compatibility library, libcompat.**

The **rexec**() function looks up the host **ahost* using gethostbyname(3), returning −1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's .netrc file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; the call getservbyname("exec", "tcp") (see getservent(3)) will return a pointer to a structure, which contains the necessary port. The protocol for connection is described in detail in rexecd(8).

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as *stdin* and *stdout*. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. The diagnostic information returned does not include remote authorization failure, as the secondary connection is set up after authorization has been verified. If *fd2p* is 0, then the *stderr* (unit 2 of the remote command) will be made the same as the *stdout* and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

rcmd(3), rexecd(8)

HISTORY

The **rexec**() function appeared in 4.2BSD.

NAME

rindex — locate character in string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <strings.h>

char *
rindex(const char *s, int c);
```

DESCRIPTION

The **rindex**() function locates the last character matching *c* (converted to a *char*) in the nul-terminated string *s*.

RETURN VALUES

A pointer to the character is returned if it is found; otherwise NULL is returned. If *c* is '\0', **rindex**() locates the terminating '\0'.

SEE ALSO

index(3), memchr(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3)

HISTORY

A **rindex**() function appeared in Version 6 AT&T UNIX.

NAME

rint, **rintf** — round to integral value in floating-point format

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
rint(double x);

float
rintf(float x);
```

DESCRIPTION

The **rint**() function returns the integral value (represented as a double precision number) nearest to *x* according to the prevailing rounding mode.

SEE ALSO

abs(3), ceil(3), fabs(3), floor(3), ieee(3), math(3)

HISTORY

A **rint**() function appeared in Version 6 AT&T UNIX.

NAME

RMD160Init, **RMD160Update**, **RMD160Final**, **RMD160Transform**, **RMD160End**, **RMD160File**, **RMD160Data** — calculate the ‘‘RIPEMD-160’’ message digest

SYNOPSIS

```
#include <sys/types.h>
#include <rmd160.h>

void
RMD160Init(RMD160_CTX *context);

void
RMD160Update(RMD160_CTX *context, const u_char *data, u_int nbytes);

void
RMD160Final(u_char digest[20], RMD160_CTX *context);

void
RMD160Transform(uint32_t state[5], const uint32_t block[16]);

char *
RMD160End(RMD160_CTX *context, char *buf);

char *
RMD160File(char *filename, char *buf);

char *
RMD160Data(u_char *data, size_t len, char *buf);
```

DESCRIPTION

The RMD160 functions implement the 160-bit RIPE message digest hash algorithm (RMD-160). RMD-160 is used to generate a condensed representation of a message called a message digest. The algorithm takes a message less than 2^64 bits as input and produces a 160-bit digest suitable for use as a digital signature.

The RMD160 functions are considered to be more secure than the `md4`(3) and `md5`(3) functions and at least as secure as the `sha1`(3) function. All share a similar interface.

The **RMD160Init**() function initializes a RMD160_CTX *context* for use with **RMD160Update**(), and **RMD160Final**(). The **RMD160Update**() function adds *data* of length *nbytes* to the RMD160_CTX specified by *context*. **RMD160Final**() is called when all data has been added via **RMD160Update**() and stores a message digest in the *digest* parameter. When a null pointer is passed to **RMD160Final**() as first argument only the final padding will be applied and the current context can still be used with **RMD160Update**().

The **RMD160Transform**() function is used by **RMD160Update**() to hash 512-bit blocks and forms the core of the algorithm. Most programs should use the interface provided by **RMD160Init**(), **RMD160Update**() and **RMD160Final**() instead of calling **RMD160Transform**() directly.

The **RMD160End**() function is a front end for **RMD160Final**() which converts the digest into an ASCII representation of the 160 bit digest in hexadecimal.

The **RMD160File**() function calculates the digest for a file and returns the result via **RMD160End**(). If **RMD160File**() is unable to open the file a NULL pointer is returned.

The **RMD160Data**() function calculates the digest of an arbitrary string and returns the result via **RMD160End**().

For each of the **RMD160End**(), **RMD160File**(), and **RMD160Data**() functions the *buf* parameter should either be a string of at least 41 characters in size or a NULL pointer. In the latter case, space will be dynamically allocated via `malloc`(3) and should be freed using `free`(3) when it is no longer needed.

EXAMPLES

The follow code fragment will calculate the digest for the string "abc" which is ‘‘0x8eb208f7e05d987a9b044a8e98c6b087f15a0bfc’’.

```
RMD160_CTX rmd;
u_char results[20];
char *buf;
int n;

buf = "abc";
n = strlen(buf);
RMD160Init(&rmd);
RMD160Update(&rmd, (u_char *)buf, n);
RMD160Final(results, &rmd);

/* Print the digest as one long hex value */
printf("0x");
for (n = 0; n < 20; n++)
        printf("%02x", results[n]);
putchar('\n');
```

Alternately, the helper functions could be used in the following way:

```
RMD160_CTX rmd;
u_char output[41];
char *buf = "abc";

printf("0x%s\n", RMD160Data(buf, strlen(buf), output));
```

SEE ALSO

`rmd160`(1), `md4`(3), `md5`(3), `sha1`(3)

H. Dobbertin, A. Bosselaers, B. Preneel, *RIPEMD-160, a strengthened version of RIPEMD*.

Information technology - Security techniques - Hash-functions - Part 3: Dedicated hash-functions, ISO/IEC 10118-3.

H. Dobbertin, A. Bosselaers, B. Preneel, "The RIPEMD-160 cryptographic hash function", *Dr. Dobb’s Journal*, Vol. 22, No. 1, pp. 24-28, January 1997.

HISTORY

The RMD-160 functions appeared in OpenBSD 2.1.

AUTHORS

This implementation of RMD-160 was written by Antoon Bosselaers.

The **RMD160End**(), **RMD160File**(), and **RMD160Data**() helper functions are derived from code written by Poul-Henning Kamp.

BUGS

If a message digest is to be copied to a multi-byte type (ie: an array of five 32-bit integers) it will be necessary to perform byte swapping on little endian machines such as the i386, alpha, and VAX.

NAME

rmtops — access tape drives on remote machines

LIBRARY

Remote Magnetic Tape Library (librmt, -lrmt)

SYNOPSIS

```
#include <rmt.h>
#include <sys/stat.h>

int
isrmt(int fd);

int
rmtaccess(char *file, int mode);

int
rmtclose(int fd);

int
rmtcreat(char *file, int mode);

int
rmtdup(int fd);

int
rmtfcntl(int fd, int cmd, int arg);

int
rmtfstat(int fd, struct stat *buf);

int
rmtioctl(int fd, int request, char *argp);

int
rmtisatty(int fd);

long
rmtlseek(int fd, long offset, int whence);

int
rmtlstat(char *file, struct stat *buf);

int
rmtopen(char *file, int flags, int mode);

int
rmtread(int fd, char *buf, int nbytes);

int
rmtstat(char *file, struct stat *buf);

int
rmtwrite(int fd, char *buf, int nbytes);
```

DESCRIPTION

The **rmtops** library provides a simple means of transparently accessing tape drives on remote machines via rsh(1) and rmt(8). These routines are used like their corresponding system calls, but allow the user to open up a tape drive on a remote system on which he or she has an account and the appropriate remote permis-

sions.

A remote tape drive file name has the form

[user@]hostname:/dev/???

where *system* is the remote system, */dev/???* is the particular drive on the remote system (raw, blocked, rewinding, non-rewinding, etc.), and the optional *user* is the login name to be used on the remote system, if different from the current user's login name.

For transparency, the user should include the file `<rmt.h>`, which has the following defines in it:

```
#define access rmtaccess
#define close  rmtclose
#define creat  rmtcreat
#define dup    rmtdup
#define fcntl  rmtfcntl
#define fstat  rmtfstat
#define ioctl  rmtioctl
#define isatty rmtisatty
#define lseek  rmtlseek
#define lstat  rmtlstat
#define open   rmtopen
#define read   rmtread
#define stat   rmtstat
#define write  rmtwrite
```

This allows the programmer to use `open`(2), `close`(2), `read`(2), `write`(2), etc. in their normal fashion, with the **`rmtops`** routines taking care of differentiating between local and remote files. This file should be included *before* including the file `<sys/stat.h>`, since it redefines the identifier ''stat'' which is used to declare objects of type *struct stat*.

The routines differentiate between local and remote file descriptors by adding a bias (currently 128) to the file descriptor of the pipe. The programmer, if he or she must know if a file is remote, should use **`isrmt`**().

ENVIRONMENT

The RCMD_CMD environment variable can be set to the name or pathname of a program to use, instead of `/usr/bin/rsh`, and must have the same calling conventions as `rsh`(1).

FILES

`/usr/lib/librmt.a` remote tape library

DIAGNOSTICS

Several of these routines will return −1 and set *errno* to EOPNOTSUPP, if they are given a remote file name or a file descriptor on an open remote file (e.g., **`rmtdup`**()).

SEE ALSO

`rcp`(1), `rsh`(1), `rmt`(8)

And the appropriate system calls in section 2.

AUTHORS

Jeff Lee wrote the original routines for accessing tape drives via `rmt`(8).

Fred Fish redid them into a general purpose library.

Arnold Robbins added the ability to specify a user name on the remote system, the `<rmt.h>` include file, this man page, cleaned up the library a little, and made the appropriate changes for 4.3BSD.

Dan Kegel contributed the code to use the `rexec`(3) library routine.

BUGS

There is no way to use remote tape drives with `stdio`(3), short of recompiling it entirely to use these routines.

The `rmt`(8) protocol is not very capable. In particular, it relies on TCP/IP sockets for error free transmission, and does no data validation of its own.

NAME

round, **roundf** — round to nearest integral value

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
round(double x);

float
roundf(float x);
```

DESCRIPTION

The **round**() and **roundf**() functions return the nearest integral value to *x*; if *x* lies halfway between two integral values, then these functions return the integral value with the larger absolute value (i.e., they round away from zero).

SEE ALSO

ceil(3), floor(3), ieee(3), math(3), rint(3), trunc(3)

STANDARDS

The **round**() and **roundf**() functions conform to ISO/IEC 9899:1999 (“ISO C99”).

HISTORY

The **round**() and **roundf**() functions appeared in NetBSD 2.0.

NAME

rpc — library routines for remote procedure calls

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <rpc/rpc.h>
#include <netconfig.h>
```

DESCRIPTION

These routines allow C language programs to make procedure calls on other machines across a network. First, the client sends a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

All RPC routines require the header ⟨rpc/rpc.h⟩. Routines that take a *netconfig* structure also require that ⟨netconfig.h⟩ be included.

Nettype

Some of the high-level RPC interface routines take a *nettype* string as one of the parameters (for example, **clnt_create**(), **svc_create**(), **rpc_reg**(), **rpc_call**(). This string defines a class of transports which can be used for a particular application.

nettype can be one of the following:

netpath	Choose from the transports which have been indicated by their token names in the NETPATH environment variable. If NETPATH is unset or NULL , it defaults to *visible*. *netpath* is the default *nettype*.
visible	Choose the transports which have the visible flag (v) set in the /etc/netconfig file.
circuit_v	This is same as *visible* except that it chooses only the connection oriented transports (semantics *tpi_cots* or *tpi_cots_ord*) from the entries in the /etc/netconfig file.
datagram_v	This is same as *visible* except that it chooses only the connectionless datagram transports (semantics *tpi_clts*) from the entries in the /etc/netconfig file.
circuit_n	This is same as *netpath* except that it chooses only the connection oriented datagram transports (semantics *tpi_cots* or *tpi_cots_ord*).
datagram_n	This is same as *netpath* except that it chooses only the connectionless datagram transports (semantics *tpi_clts*).
udp	This refers to Internet UDP, both version 4 and 6.
tcp	This refers to Internet TCP, both version 4 and 6.

If is NULL, it defaults to *netpath*. The transports are tried in left to right order in the NETPATH variable or in top to down order in the /etc/netconfig file.

Derived Types

The derived types used in the RPC interfaces are defined as follows:

```
typedef uint32_t rpcprog_t;
typedef uint32_t rpcvers_t;
typedef uint32_t rpcproc_t;
typedef uint32_t rpcprot_t;
```

```
typedef uint32_t rpcport_t;
typedef   int32_t rpc_inline_t;
```

Data Structures

Some of the data structures used by the RPC package are shown below.

The AUTH Structure

```
/*
 * Authentication info. Opaque to client.
 */
struct opaque_auth {
    enum_t    oa_flavor;    /* flavor of auth */
    caddr_t    oa_base;    /* address of more auth stuff */
    u_int    oa_length;    /* not to exceed MAX_AUTH_BYTES */
};

/*
 * Auth handle, interface to client side authenticators.
 */
typedef struct {
    struct    opaque_auth    ah_cred;
    struct    opaque_auth    ah_verf;
    struct auth_ops {
        void    (*ah_nextverf)();
        int    (*ah_marshal)();    /* nextverf & serialize */
        int    (*ah_validate)();    /* validate verifier */
        int    (*ah_refresh)();    /* refresh credentials */
        void    (*ah_destroy)();    /* destroy this structure */
    } *ah_ops;
    caddr_t ah_private;
} AUTH;
```

The CLIENT Structure

```
/*
 * Client rpc handle.
 * Created by individual implementations.
 * Client is responsible for initializing auth.
 */

typedef struct {
    AUTH    *cl_auth;    /* authenticator */
    struct clnt_ops {
        enum clnt_stat    (*cl_call)();    /* call remote procedure */
        void    (*cl_abort)();        /* abort a call */
        void    (*cl_geterr)();        /* get specific error code */
        bool_t    (*cl_freeres)();    /* frees results */
        void    (*cl_destroy)();    /* destroy this structure */
        bool_t    (*cl_control)();    /* the ioctl() of rpc */
    } *cl_ops;
    caddr_t    cl_private;    /* private stuff */
    char    *cl_netid;    /* network identifier */
    char    *cl_tp;        /* device name */
```

```
} CLIENT;
```

The SVCXPRT structure

```
enum xprt_stat {
    XPRT_DIED,
    XPRT_MOREREQS,
    XPRT_IDLE
};

/*
 * Server side transport handle
 */
typedef struct {
    int    xp_fd;    /* file descriptor for the server handle */
    u_short    xp_port;    /* obsolete */
    const struct xp_ops {
        bool_t    (*xp_recv)();    /* receive incoming requests */
        enum xprt_stat    (*xp_stat)();    /* get transport status */
        bool_t    (*xp_getargs)();    /* get arguments */
        bool_t    (*xp_reply)();      /* send reply */
        bool_t    (*xp_freeargs)(); /* free mem allocated for args */
        void    (*xp_destroy)();    /* destroy this struct */
    } *xp_ops;
    int    xp_addrlen;    /* length of remote addr.  Obsolete */
    struct sockaddr_in    xp_raddr; /* Obsolete */
    const struct xp_ops2 {
        bool_t    (*xp_control)();    /* catch-all function */
    } *xp_ops2;
    char    *xp_tp;    /* transport provider device name */
    char    *xp_netid;    /* network identifier */
    struct netbuf    xp_ltaddr;    /* local transport address */
    struct netbuf    xp_rtaddr;    /* remote transport address */
    struct opaque_auth    xp_verf;    /* raw response verifier */
    caddr_t    xp_p1;    /* private: for use by svc ops */
    caddr_t    xp_p2;    /* private: for use by svc ops */
    caddr_t    xp_p3;    /* private: for use by svc lib */
    int    xp_type    /* transport type */
} SVCXPRT;
```

The svc_req structure

```
struct svc_req {
    rpcprog_t    rq_prog;    /* service program number */
    rpcvers_t    rq_vers;    /* service protocol version */
    rpcproc_t    rq_proc;    /* the desired procedure */
    struct opaque_auth    rq_cred;    /* raw creds from the wire */
    caddr_t    rq_clntcred;    /* read only cooked cred */
    SVCXPRT    *rq_xprt;    /* associated transport */
};
```

The XDR structure

```
/*
 * XDR operations.
 * XDR_ENCODE causes the type to be encoded into the stream.
 * XDR_DECODE causes the type to be extracted from the stream.
 * XDR_FREE can be used to release the space allocated by an XDR_DECODE
 * request.
 */
enum xdr_op {
    XDR_ENCODE=0,
    XDR_DECODE=1,
    XDR_FREE=2
};
/*
 * This is the number of bytes per unit of external data.
 */
#define BYTES_PER_XDR_UNIT    (4)
#define RNDUP(x)  ((((x) + BYTES_PER_XDR_UNIT - 1) /
                   BYTES_PER_XDR_UNIT) \ * BYTES_PER_XDR_UNIT)

/*
 * A xdrproc_t exists for each data type which is to be encoded or
 * decoded.  The second argument to the xdrproc_t is a pointer to
 * an opaque pointer.  The opaque pointer generally points to a
 * structure of the data type to be decoded.  If this points to 0,
 * then the type routines should allocate dynamic storage of the
 * appropriate size and return it.
 * bool_t  (*xdrproc_t)(XDR *, caddr_t *);
 */
typedef  bool_t (*xdrproc_t)();

/*
 * The XDR handle.
 * Contains operation which is being applied to the stream,
 * an operations vector for the particular implementation
 */
typedef struct {
    enum xdr_op    x_op;    /* operation; fast additional param */
    struct xdr_ops {
        bool_t    (*x_getlong)();    /* get a long from underlying stream */
        bool_t    (*x_putlong)();    /* put a long to underlying stream */
        bool_t    (*x_getbytes)(); /* get bytes from underlying stream */
        bool_t    (*x_putbytes)(); /* put bytes to underlying stream */
        u_int    (*x_getpostn)(); /* returns bytes off from beginning */
        bool_t    (*x_setpostn)(); /* lets you reposition the stream */
        long *    (*x_inline)();    /* buf quick ptr to buffered data */
        void    (*x_destroy)();    /* free privates of this xdr_stream */
    } *x_ops;
    caddr_t    x_public;    /* users' data */
    caddr_t    x_private;    /* pointer to private data */
    caddr_t    x_base;    /* private used for position info */
    int    x_handy;    /* extra private word */
```

```
} XDR;

/*
 * The netbuf structure. This structure is defined in <xti.h> on SysV
 * systems, but NetBSD does not use XTI.
 *
 * Usually, buf will point to a struct sockaddr, and len and maxlen
 * will contain the length and maximum length of that socket address,
 * respectively.
 */
struct netbuf {
        unsigned int maxlen;
        unsigned int len;
        void *buf;
};

/*
 * The format of the address and options arguments of the XTI t_bind call.
 * Only provided for compatibility, it should not be used other than
 * as an argument to svc_tli_create().
 */

struct t_bind {
        struct netbuf   addr;
        unsigned int    qlen;
};
```

Index to Routines

The following table lists RPC routines and the manual reference pages on which they are described:

RPC Routine	*Manual Reference Page*
auth_destroy()	rpc_clnt_auth(3),
authdes_create()	rpc_soc(3),
authnone_create()	rpc_clnt_auth(3),
authsys_create()	rpc_clnt_auth(3),
authsys_create_default()	rpc_clnt_auth(3),
authunix_create()	rpc_soc(3),
authunix_create_default()	rpc_soc(3),
callrpc()	rpc_soc(3),
clnt_broadcast()	rpc_soc(3),
clnt_call()	rpc_clnt_calls(3),
clnt_control()	rpc_clnt_create(3),
clnt_create()	rpc_clnt_create(3),
clnt_destroy()	rpc_clnt_create(3),
clnt_dg_create()	rpc_clnt_create(3),
clnt_freeres()	rpc_clnt_calls(3),
clnt_geterr()	rpc_clnt_calls(3),
clnt_pcreateerror()	rpc_clnt_create(3),
clnt_perrno()	rpc_clnt_calls(3),

clnt_perror() rpc_clnt_calls(3),
clnt_raw_create() rpc_clnt_create(3),
clnt_spcreateerror() rpc_clnt_create(3),
clnt_sperrno() rpc_clnt_calls(3),
clnt_sperror() rpc_clnt_calls(3),
clnt_tli_create() rpc_clnt_create(3),
clnt_tp_create() rpc_clnt_create(3),
clnt_udpcreate() rpc_soc(3),
clnt_vc_create() rpc_clnt_create(3),
clntraw_create() rpc_soc(3),
clnttcp_create() rpc_soc(3),
clntudp_bufcreate() rpc_soc(3),
get_myaddress() rpc_soc(3),
pmap_getmaps() rpc_soc(3),
pmap_getport() rpc_soc(3),
pmap_rmtcall() rpc_soc(3),
pmap_set() rpc_soc(3),
pmap_unset() rpc_soc(3),
registerrpc() rpc_soc(3),
rpc_broadcast() rpc_clnt_calls(3),
rpc_broadcast_exp() rpc_clnt_calls(3),
rpc_call() rpc_clnt_calls(3),
rpc_reg() rpc_svc_calls(3),
svc_create() rpc_svc_create(3),
svc_destroy() rpc_svc_create(3),
svc_dg_create() rpc_svc_create(3),
svc_dg_enablecache() rpc_svc_calls(3),
svc_fd_create() rpc_svc_create(3),
svc_fds() rpc_soc(3),
svc_freeargs() rpc_svc_reg(3),
svc_getargs() rpc_svc_reg(3),
svc_getcaller() rpc_soc(3),
svc_getreq() rpc_soc(3),
svc_getreqset() rpc_svc_calls(3),
svc_getrpccaller() rpc_svc_calls(3),
svc_kerb_reg() kerberos_rpc(3),
svc_raw_create() rpc_svc_create(3),
svc_reg() rpc_svc_calls(3),
svc_register() rpc_soc(3),
svc_run() rpc_svc_reg(3),
svc_sendreply() rpc_svc_reg(3),
svc_tli_create() rpc_svc_create(3),
svc_tp_create() rpc_svc_create(3),
svc_unreg() rpc_svc_calls(3),
svc_unregister() rpc_soc(3),
svc_vc_create() rpc_svc_create(3),
svcerr_auth() rpc_svc_err(3),
svcerr_decode() rpc_svc_err(3),
svcerr_noproc() rpc_svc_err(3),

svcerr_noprog() rpc_svc_err(3),
svcerr_progvers() rpc_svc_err(3),
svcerr_systemerr() rpc_svc_err(3),
svcerr_weakauth() rpc_svc_err(3),
svcfd_create() rpc_soc(3),
svcraw_create() rpc_soc(3),
svctcp_create() rpc_soc(3),
svcudp_bufcreate() rpc_soc(3),
svcudp_create() rpc_soc(3),
xdr_accepted_reply() rpc_xdr(3),
xdr_authsys_parms() rpc_xdr(3),
xdr_authunix_parms() rpc_soc(3),
xdr_callhdr() rpc_xdr(3),
xdr_callmsg() rpc_xdr(3),
xdr_opaque_auth() rpc_xdr(3),
xdr_rejected_reply() rpc_xdr(3),
xdr_replymsg() rpc_xdr(3),
xprt_register() rpc_svc_calls(3),
xprt_unregister() rpc_svc_calls(3),

FILES

/etc/netconfig

SEE ALSO

getnetconfig(3), getnetpath(3), rpc_clnt_auth(3), rpc_clnt_calls(3), rpc_clnt_create(3), rpc_svc_calls(3), rpc_svc_create(3), rpc_svc_err(3), rpc_svc_reg(3), rpc_xdr(3), rpcbind(3), xdr(3), netconfig(5)

NAME

auth_destroy, **authnone_create**, **authsys_create**, **authsys_create_default** — library routines for client side remote procedure call authentication

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <rpc/rpc.h>

void
auth_destroy(AUTH *auth);

AUTH *
authnone_create(void);

AUTH *
authsys_create(const char *host, const uid_t uid, const gid_t gid,
    const int len, const gid_t *aup_gids);

AUTH *
authsys_create_default(void);
```

DESCRIPTION

These routines are part of the RPC library that allows C language programs to make procedure calls on other machines across the network, with desired authentication.

These routines are normally called after creating the `CLIENT` handle. The *cl_auth* field of the `CLIENT` structure should be initialized by the `AUTH` structure returned by some of the following routines. The client's authentication information is passed to the server when the RPC call is made.

Only the `NULL` and the `SYS` style of authentication is discussed here.

ROUTINES

auth_destroy() A function macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of **auth**() is undefined after calling **auth_destroy**().

authnone_create() Create and return an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default authentication used by RPC.

authsys_create() Create and return an RPC authentication handle that contains `AUTH_SYS` authentication information. The parameter *host* is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *len* and *aup_gids* refer to a counted array of groups to which the user belongs.

authsys_create_default() Call **authsys_create**() with the appropriate parameters.

SEE ALSO

`rpc`(3), `rpc_clnt_calls`(3), `rpc_clnt_create`(3)

NAME

rpc_clnt_calls, **clnt_call**, **clnt_freeres**, **clnt_geterr**, **clnt_perrno**, **clnt_perror**, **clnt_sperrno**, **clnt_sperror**, **rpc_broadcast**, **rpc_broadcast_exp**, **rpc_call** — library routines for client side calls

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <rpc/rpc.h>

enum clnt_stat
clnt_call(CLIENT *clnt, const rpcproc_t procnum, const xdrproc_t inproc,
    const char *in, const xdrproc_t outproc, caddr_t out,
    const struct timeval tout);

bool_t
clnt_freeres(CLIENT *clnt, const xdrproc_t outproc, caddr_t out);

void
clnt_geterr(const CLIENT * clnt, struct rpc_err * errp);

void
clnt_perrno(const enum clnt_stat stat);

void
clnt_perror(const CLIENT * clnt, const char *s);

char *
clnt_sperrno(const enum clnt_stat stat);

char *
clnt_sperror(const CLIENT *clnt, const char * s);

enum clnt_stat
rpc_broadcast(const rpcprog_t prognum, const rpcvers_t versnum,
    const rpcproc_t procnum, const xdrproc_t inproc, const char *in,
    const xdrproc_t outproc, caddr_t out, const resultproc_t eachresult,
    const char *nettype);

enum clnt_stat
rpc_broadcast_exp(rpcprog_t prognum, const rpcvers_t versnum,
    const rpcproc_t procnum, const xdrproc_t xargs,
    caddr_t argsp, const xdrproc_t xresults, caddr_t resultsp,
    const int inittime, const int waittime, const resultproc_t eachresult,
    const char * nettype);

enum clnt_stat
rpc_call(const char *host, const rpcprog_t prognum,
    const rpcvers_t versnum,
    const rpcproc_t procnum, const xdrproc_t inproc, const char *in,
    const xdrproc_t outproc, char *out, const char *nettype);
```

DESCRIPTION

RPC library routines allow C language programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

The **clnt_call**(), **rpc_call**(), and **rpc_broadcast**() routines handle the client side of the procedure call. The remaining routines deal with error handling in the case of errors.

Some of the routines take a CLIENT handle as one of the parameters. A CLIENT handle can be created by an RPC creation routine such as **clnt_create**() (see rpc_clnt_create(3)).

These routines are safe for use in multithreaded applications. CLIENT handles can be shared between threads, however in this implementation requests by different threads are serialized (that is, the first request will receive its results before the second request is sent).

ROUTINES

See rpc(3) for the definition of the CLIENT data structure.

clnt_call()
: A function macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as **clnt_create**() (see rpc_clnt_create(3)). The parameter **inproc**() is the XDR function used to encode the procedure's parameters, and **outproc**() is the XDR function used to decode the procedure's results; **in**() is the address of the procedure's argument(s), and **out**() is the address of where to place the result(s). **tout**() is the time allowed for results to be returned, which is overridden by a time-out set explicitly through **clnt_control**(), see rpc_clnt_create(3). If the remote call succeeds, the status returned is RPC_SUCCESS, otherwise an appropriate status is returned.

clnt_freeres()
: A function macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter *out* is the address of the results, and *outproc* is the XDR routine describing the results. This routine returns 1 if the results were successfully freed, and 0 otherwise.

clnt_geterr()
: A function macro that copies the error structure out of the client handle to the structure at address *errp*.

clnt_perrno()
: Print a message to standard error corresponding to the condition indicated by *stat*. A newline is appended. Normally used after a procedure call fails for a routine for which a client handle is not needed, for instance **rpc_call**().

clnt_perror()
: Print a message to the standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. A newline is appended. Normally used after a remote procedure call fails for a routine which requires a client handle, for instance **clnt_call**().

clnt_sperrno()
: Take the same arguments as **clnt_perrno**(), but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which contains the message. **clnt_sperrno**() is normally used instead of **clnt_perrno**() when the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with **printf**() (see printf(3)), or if a message format different than that supported by **clnt_perrno**() is to be used. Note: unlike **clnt_sperror**() and **clnt_spcreaterror**() (see rpc_clnt_create(3)), **clnt_sperrno**() does not return pointer to static data so the result will not get overwritten on each call.

clnt_sperror()
: Like **clnt_perror**(), except that (like **clnt_sperrno**()) it returns a string instead of printing to standard error. However, **clnt_sperror**() does not append a newline at the end of the mes-

sage. Warning: returns pointer to a buffer that is overwritten on each call.

rpc_broadcast()
Like **rpc_call**(), except the call message is broadcast to all the connectionless transports specified by *nettype*. If *nettype* is NULL, it defaults to "netpath". Each time it receives a response, this routine calls **eachresult**(), whose form is: *bool_t* **eachresult**(*caddr_t out*, *const struct netbuf * addr*, *const struct netconfig * netconf*) where *out* is the same as *out* passed to **rpc_broadcast**(), except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results, and *netconf* is the netconfig structure of the transport on which the remote server responded. If **eachresult**() returns 0, **rpc_broadcast**() waits for more replies; otherwise it returns with appropriate status. Warning: broadcast file descriptors are limited in size to the maximum transfer size of that transport. For Ethernet, this value is 1500 bytes. **rpc_broadcast**() uses AUTH_SYS credentials by default (see rpc_clnt_auth(3)).

rpc_broadcast_exp()
Like **rpc_broadcast**(), except that the initial timeout, *inittime* and the maximum timeout, *waittime* are specified in milliseconds. *inittime* is the initial time that **rpc_broadcast_exp**() waits before resending the request. After the first resend, the re-transmission interval increases exponentially until it exceeds *waittime*.

rpc_call()
Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results; *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s). *nettype* can be any of the values listed on rpc(3). This routine returns RPC_SUCCESS if it succeeds, or an appropriate status is returned. Use the **clnt_perrno**() routine to translate failure status into error messages. Warning: **rpc_call**() uses the first available transport belonging to the class *nettype*, on which it can create a connection. You do not have control of timeouts or authentication using this routine.

SEE ALSO
printf(3), rpc(3), rpc_clnt_auth(3), rpc_clnt_create(3)

NAME

rpc_clnt_create, **clnt_control**, **clnt_create**, **clnt_create_vers**, **clnt_destroy**, **clnt_dg_create**, **clnt_pcreateerror**, **clnt_raw_create**, **clnt_spcreateerror**, **clnt_tli_create**, **clnt_tp_create**, **clnt_vc_create**, **rpc_createerr** — library routines for dealing with creation and manipulation of CLIENT handles

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <rpc/rpc.h>

bool_t
clnt_control(CLIENT *clnt, const u_int req, char *info);

CLIENT *
clnt_create(const char * host, const rpcprog_t prognum,
    const rpcvers_t versnum, const char *nettype);

CLIENT *
clnt_create_vers(const char * host, const rpcprog_t prognum,
    rpcvers_t *vers_outp, const rpcvers_t vers_low,
    const rpcvers_t vers_high, char *nettype);

void
clnt_destroy(CLIENT *, clnt);

CLIENT *
clnt_dg_create(const int fildes, const struct netbuf *svcaddr,
    const rpcprog_t prognum, const rpcvers_t versnum, const u_int sendsz,
    const u_int recvsz);

void
clnt_pcreateerror(const char *s);

char *
clnt_spcreateerror(const char *s);

CLIENT *
clnt_raw_create(const rpcprog_t prognum, const rpcvers_t versnum);

CLIENT *
clnt_tli_create(const int fildes, const struct netconfig *netconf,
    const struct netbuf *svcaddr, const rpcprog_t prognum,
    const rpcvers_t versnum, const u_int sendsz, const u_int recvsz);

CLIENT *
clnt_tp_create(const char * host, const rpcprog_t prognum,
    const rpcvers_t versnum, const struct netconfig *netconf);

CLIENT *
clnt_vc_create(const int fildes, const struct netbuf *svcaddr,
    const rpcprog_t prognum, const rpcvers_t versnum, const u_int sendsz,
    const u_int recvsz);
```

DESCRIPTION

RPC library routines allow C language programs to make procedure calls on other machines across the network. First a CLIENT handle is created and then the client calls a procedure to send a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends a reply.

ROUTINES

clnt_control()

A function macro to change or retrieve various information about a client object. *req* indicates the type of operation, and *info* is a pointer to the information. For both connectionless and connection-oriented transports, the supported values of *req* and their argument types and what they do are:

CLSET_TIMEOUT	struct timeval *	set total timeout
CLGET_TIMEOUT	struct timeval *	get total timeout

Note: if you set the timeout using **clnt_control**(), the timeout argument passed by **clnt_call**() is ignored in all subsequent calls.

Note: If you set the timeout value to 0 **clnt_control**() immediately returns an error (RPC_TIMEDOUT). Set the timeout parameter to 0 for batching calls.

CLGET_SVC_ADDR	struct netbuf *	get servers address
CLGET_FD	int *	get fd from handle
CLSET_FD_CLOSE	void	close fd on destroy
CLSET_FD_NCLOSE	void	don't close fd on destroy
CLGET_VERS	unsigned long *	get RPC program version
CLSET_VERS	unsigned long *	set RPC program version
CLGET_XID	unsigned long *	get XID of previous call
CLSET_XID	unsigned long *	set XID of next call

The following operations are valid for connectionless transports only:

CLSET_RETRY_TIMEOUT	struct timeval *	set the retry timeout
CLGET_RETRY_TIMEOUT	struct timeval *	get the retry timeout

The retry timeout is the time that RPC waits for the server to reply before retransmitting the request. **clnt_control**() returns TRUE on success and FALSE on failure.

clnt_create()

Generic client creation routine for program *prognum* and version *versnum*. *host* identifies the name of the remote host where the server is located. *nettype* indicates the class of transport protocol to use. The transports are tried in left to right order in NETPATH environment variable or in top to bottom order in the netconfig database. **clnt_create**() tries all the transports of the *nettype* class available from the NETPATH environment variable and the netconfig database, and chooses the first successful one. A default timeout is set and can be modified using **clnt_control**(). This routine returns NULL if it fails. The **clnt_pcreateerror**() routine can be used to print the reason for failure.

Note: **clnt_create**() returns a valid client handle even if the particular version number supplied to **clnt_create**() is not registered with the rpcbind(8) service. This mismatch will be discovered by a **clnt_call**() later (see rpc_clnt_calls(3)).

clnt_create_vers()

Generic client creation routine which is similar to **clnt_create**() but which also checks for the version availability. *host* identifies the name of the remote host where the server is located. *nettype* indicates the class transport protocols to be used. If the routine is successful it returns

a client handle created for the highest version between *vers_low* and *vers_high* that is supported by the server. *vers_outp* is set to this value. That is, after a successful return *vers_low* ≤ ∗*vers_outp* ≤ *vers_high*. If no version between *vers_low* and *vers_high* is supported by the server then the routine fails and returns NULL. A default timeout is set and can be modified using **clnt_control**(). This routine returns NULL if it fails. The **clnt_pcreateerror**() routine can be used to print the reason for failure. Note: **clnt_create**() returns a valid client handle even if the particular version number supplied to **clnt_create**() is not registered with the rpcbind(8) service. This mismatch will be discovered by a **clnt_call**() later (see rpc_clnt_calls(3)). However, **clnt_create_vers**() does this for you and returns a valid handle only if a version within the range supplied is supported by the server.

clnt_destroy()
A function macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling **clnt_destroy**(). If the RPC library opened the associated file descriptor, or CLSET_FD_CLOSE was set using **clnt_control**(), the file descriptor will be closed. The caller should call **auth_destroy**(*clnt->cl_auth*) (before calling **clnt_destroy**()) to destroy the associated AUTH structure (see rpc_clnt_auth(3)).

clnt_dg_create()
This routine creates an RPC client for the remote program *prognum* and version *versnum*; the client uses a connectionless transport. The remote program is located at address *svcaddr*. The parameter *fildes* is an open and bound file descriptor. This routine will resend the call message in intervals of 15 seconds until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call**() (see **clnt_call**() in rpc_clnt_calls(3)). The retry time out and the total time out periods can be changed using **clnt_control**(). The user may set the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. This routine returns NULL if it fails.

clnt_pcreateerror()
Print a message to standard error indicating why a client RPC handle could not be created. The message is prepended with the string *s* and a colon, and appended with a newline.

clnt_spcreateerror()
Like **clnt_pcreateerror**(), except that it returns a string instead of printing to the standard error. A newline is not appended to the message in this case. Warning: returns a pointer to a buffer that is overwritten on each call.

clnt_raw_create()
This routine creates an RPC client handle for the remote program *prognum* and version *versnum*. The transport used to pass messages to the service is a buffer within the process's address space, so the corresponding RPC server should live in the same address space; (see **svc_raw_create**() in rpc_svc_create(3)). This allows simulation of RPC and measurement of RPC overheads, such as round trip times, without any kernel or networking interference. This routine returns NULL if it fails. **clnt_raw_create**() should be called after **svc_raw_create**().

clnt_tli_create()
This routine creates an RPC client handle for the remote program *prognum* and version *versnum*. The remote program is located at address *svcaddr*. If *svcaddr* is NULL and it is connection-oriented, it is assumed that the file descriptor is connected. For connectionless transports, if *svcaddr* is RPC_UNKNOWNADDR error is set. *fildes* is a file descriptor which may be open, bound and connected. If it is RPC_ANYFD, it opens a file descriptor on the transport

specified by *netconf*. If *fildes* is RPC_ANYFD and *netconf* is NULL, a RPC_UNKNOWNPROTO error is set. If *fildes* is unbound, then it will attempt to bind the descriptor. The user may specify the size of the buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. Depending upon the type of the transport (connection-oriented or connectionless), **clnt_tli_create**() calls appropriate client creation routines. This routine returns NULL if it fails. The **clnt_pcreateerror**() routine can be used to print the reason for failure. The remote rpcbind service (see rpcbind(8)) is not consulted for the address of the remote service.

clnt_tp_create()
Like **clnt_create**() except **clnt_tp_create**() tries only one transport specified through *netconf*. **clnt_tp_create**() creates a client handle for the program *prognum*, the version *versnum*, and for the transport specified by *netconf*. Default options are set, which can be changed using **clnt_control**() calls. The remote rpcbind service on the host *host* is consulted for the address of the remote service. This routine returns NULL if it fails. The **clnt_pcreateerror**() routine can be used to print the reason for failure.

clnt_vc_create()
This routine creates an RPC client for the remote program *prognum* and version *versnum*; the client uses a connection-oriented transport. The remote program is located at address *svcaddr*. The parameter *fildes* is an open and bound file descriptor. The user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. This routine returns NULL if it fails. The address *svcaddr* should not be NULL and should point to the actual address of the remote program. **clnt_vc_create**() does not consult the remote rpcbind service for this information.

struct rpc_createerr rpc_createerr;
A global variable whose value is set by any RPC client handle creation routine that fails. It is used by the routine **clnt_pcreateerror**() to print the reason for the failure.

SEE ALSO

rpc(3), rpc_clnt_auth(3), rpc_clnt_calls(3), rpcbind(8)

NAME

rpc_soc, **auth_destroy**, **authnone_create**, **authunix_create**, **authunix_create_default**, **callrpc**, **clnt_broadcast**, **clnt_call**, **clnt_control**, **clnt_create**, **clnt_destroy**, **clnt_freeres**, **clnt_geterr**, **clnt_pcreateerror**, **clnt_perrno**, **clnt_perror**, **clnt_spcreateerror**, **clnt_sperrno**, **clnt_sperror**, **clntraw_create**, **clnttcp_create**, **clntudp_bufcreate**, **clntudp_create**, **get_myaddress**, **pmap_getmaps**, **pmap_getport**, **pmap_rmtcall**, **pmap_set**, **pmap_unset**, **registerrpc**, **rpc_createerr**, **svc_destroy**, **svc_fds**, **svc_fdset**, **svc_getargs**, **svc_getcaller**, **svc_getreg**, **svc_getregset**, **svc_register**, **svc_run**, **svc_sendreply**, **svc_unregister**, **svcerr_auth**, **svcerr_decode**, **svcerr_noproc**, **svcerr_noprog**, **svcerr_progvers**, **svcerr_systemerr**, **svcerr_weakauth**, **svcfd_create**, **svcraw_create**, **xdr_accepted_reply**, **xdr_authunix_parms**, **xdr_callhdr**, **xdr_callmsg**, **xdr_opaque_auth**, **xdr_pmap**, **xdr_pmaplist**, **xdr_rejected_reply**, **xdr_replymsg**, **xprt_register**, **xprt_unregister** — library routines for remote procedure calls

SYNOPSIS

```
#include <rpc/rpc.h>

void
auth_destroy(AUTH *auth);

AUTH *
authnone_create(void);

AUTH *
authunix_create(char *host, int uid, int gid, int len, int *aup_gids);

AUTH *
authunix_create_default(void);

int
callrpc(char *host, u_long prognum, u_long versnum, u_long procnum,
    xdrproc_t inproc, char *in, xdrproc_t outproc, char *out);

enum clnt_stat
clnt_broadcast(u_long prognum, u_long versnum, u_long procnum,
    xdrproc_t inproc, char *in, xdrproc_t outproc, char *out,
    resultproc_t eachresult);

enum clnt_stat
clnt_call(CLIENT *clnt, u_long procnum, xdrproc_t inproc, char *in,
    xdrproc_t outproc, char *out, struct timeval tout);

int
clnt_destroy(CLIENT *clnt);

CLIENT *
clnt_create(char *host, u_long prog, u_long vers, char *proto);

bool_t
clnt_control(CLIENT *cl, u_int req, char *info);

int
clnt_freeres(CLIENT *clnt, xdrproc_t outproc, char *out);

void
clnt_geterr(CLIENT *clnt, struct rpc_err errp);
```

void
clnt_pcreateerror(*char *s*);

void
clnt_perrno(*enum clnt_stat stat*);

int
clnt_perror(*CLIENT *clnt*, *char *s*);

*char **
clnt_spcreateerror(*const char *s*);

*char **
clnt_sperrno(*enum clnt_stat stat*);

*char **
clnt_sperror(*CLIENT *rpch*, *char *s*);

*CLIENT **
clntraw_create(*u_long prognum*, *u_long versnum*);

*CLIENT **
clnttcp_create(*struct sockaddr_in *addr*, *u_long prognum*, *u_long versnum*,
*int *sockp*, *u_int sendsz*, *u_int recvsz*);

*CLIENT **
clntudp_create(*struct sockaddr_in *addr*, *u_long prognum*, *u_long versnum*,
struct timeval wait, *int *sockp*);

*CLIENT **
clntudp_bufcreate(*struct sockaddr_in *addr*, *u_long prognum*, *u_long versnum*,
struct timeval wait, *int *sockp*, *unsigned int sendsize*,
unsigned int recosize);

int
get_myaddress(*struct sockaddr_in *addr*);

*struct pmaplist **
pmap_getmaps(*struct sockaddr_in *addr*);

u_short
pmap_getport(*struct sockaddr_in *addr*, *u_long prognum*, *u_long versnum*,
u_long protocol);

enum clnt_stat
pmap_rmtcall(*struct sockaddr_in *addr*,*u_long prognum*,*u_long versnum*,
u_long procnum,*xdrproc_t inproc*,*char *in*,*xdrpoc_t outproc*,*char *out*,
struct timeval tout,*u_long *portp*);

int
pmap_set(*u_long prognum*, *u_long versnum*, *int protocol*, *int port*);

int
pmap_unset(*u_long prognum*, *u_long versnum*);

int
registerrpc(*u_long prognum*, *u_long versnum*, *u_long procnum*,
*char *(*procname)()*, *xdrproc_t inproc*, *xdrproc_t outproc*);

```
struct rpc_createerr rpc_createerr;

int
svc_destroy(SVCXPRT *xprt);

fd_set svc_fdset;
int svc_fds;

int
svc_freeargs(SVCXPRT *xprt, xdrproc_t inproc, char *in);

int
svc_getargs(SVCXPRT *xprt, xdrproc_t inproc, char *in);

struct sockaddr_in *
svc_getcaller(SVCXPRT *xprt);

int
svc_getreqset(fd_set *rdfds);

int
svc_getreq(int rdfds);

int
svc_register(SVCXPRT *xprt, u_long prognum, u_long versnum,
    void (*dispatch)(), u_long protocol);

int
svc_run(void);

int
svc_sendreply(SVCXPRT *xprt, xdrproc_t outproc, char *out);

void
svc_unregister(u_long prognum, u_long versnum);

void
svcerr_auth(SVCXPRT *xprt, enum auth_stat why);

void
svcerr_decode(SVCXPRT *xprt);

void
svcerr_noproc(SVCXPRT *xprt);

void
svcerr_noprog(SVCXPRT *xprt);

void
svcerr_progvers(SVCXPRT *xprt);

void
svcerr_systemerr(SVCXPRT *xprt);

void
svcerr_weakauth(SVCXPRT *xprt);

SVCXPRT *
svcraw_create(void);
```

SVCXPRT ∗
svctcp_create(*int sock*, *u_int send_buf_size*, *u_int recv_buf_size*);

SVCXPRT ∗
svcfd_create(*int fd*, *u_int sendsize*, *u_int recvsize*);

SVCXPRT ∗
svcudp_bufcreate(*int sock*, *u_int sendsize*, *u_int recosize*);

SVCXPRT ∗
svcudp_create(*int sock*);

int
xdr_accepted_reply(*XDR* ∗*xdrs*, *struct accepted_reply* ∗*ar*);

int
xdr_authunix_parms(*XDR* ∗*xdrs*, *struct authunix_parms* ∗*aupp*);

void
xdr_callhdr(*XDR* ∗*xdrs*, *struct rpc_msg* ∗*chdr*);

int
xdr_callmsg(*XDR* ∗*xdrs*, *struct rpc_msg* ∗*cmsg*);

int
xdr_opaque_auth(*XDR* ∗*xdrs*, *struct opaque_auth* ∗*ap*);

int
xdr_pmap(*XDR* ∗*xdrs*, *struct pmap* ∗*regs*);

int
xdr_pmaplist(*XDR* ∗*xdrs*, *struct pmaplist* ∗∗*rp*);

int
xdr_rejected_reply(*XDR* ∗*xdrs*, *struct rejected_reply* ∗*rr*);

int
xdr_replymsg(*XDR* ∗*xdrs*, *struct rpc_msg* ∗*rmsg*);

void
xprt_register(*SVCXPRT* ∗*xprt*);

void
xprt_unregister(*SVCXPRT* ∗*xprt*);

DESCRIPTION

The svc and clnt functions described in this page are the old, TS-RPC interface to the XDR and RPC library, and exist for backward compatibility. The new interface is described in the pages referenced from rpc(3).

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

auth_destroy()
: A macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling **auth_destroy**().

authnone_create()
Create and returns an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default authentication used by RPC.

authunix_create()
Create and return an RPC authentication handle that contains authentication information. The parameter *host* is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group id; *len* and *aup_gids* refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

authunix_create_default()
Calls **authunix_create**() with the appropriate parameters.

callrpc()
Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. This routine returns zero if it succeeds, or the value of *enum clnt_stat* cast to an integer if it fails. The routine **clnt_perrno**() is handy for translating failure statuses into messages.

Warning: calling remote procedures with this routine uses UDP/IP as a transport; see **clntudp_create**() for restrictions. You do not have control of timeouts or authentication using this routine.

clnt_broadcast()
Like **callrpc**(), except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls **eachresult**(), whose form is *int* **eachresult**(*char *out*, *struct sockaddr_in *addr*) where *out* is the same as *out* passed to **clnt_broadcast**(), except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results. If **eachresult**() returns zero, **clnt_broadcast**() waits for more replies; otherwise it returns with appropriate status.

Warning: broadcast sockets are limited in size to the maximum transfer unit of the data link. For ethernet, this value is 1500 bytes.

clnt_call()
A macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as **clnt_create**(). The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results; *tout* is the time allowed for results to come back.

clnt_destroy()
A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling **clnt_destroy**(). If the RPC library opened the associated socket, it will close it also. Otherwise, the socket remains open.

clnt_create()
Generic client creation routine. *host* identifies the name of the remote host where the server is located. *proto* indicates which kind of transport protocol to use. The currently supported values for this field are "udp" and "tcp". Default timeouts are set, but can be modified using **clnt_control**().

Warning: Using UDP has its shortcomings. Since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

clnt_control()
A macro used to change or retrieve various information about a client object. *req* indicates the type of operation, and *info* is a pointer to the information. For both UDP and TCP the supported values of *req* and their argument types and what they do are:

CLSET_TIMEOUT *struct timeval*; set total timeout.

CLGET_TIMEOUT *struct timeval*; get total timeout.

Note: if you set the timeout using **clnt_control**(), the timeout parameter passed to **clnt_call**() will be ignored in all future calls.

CLGET_SERVER_ADDR *struct sockaddr_in*; get server's address.

The following operations are valid for UDP only:

CLSET_RETRY_TIMEOUT
struct timeval; set the retry timeout.

CLGET_RETRY_TIMEOUT
struct timeval; get the retry timeout.

The retry timeout is the time that UDP RPC waits for the server to reply before retransmitting the request.

clnt_freeres()
A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter *out* is the address of the results, and *outproc* is the XDR routine describing the results. This routine returns one if the results were successfully freed, and zero otherwise.

clnt_geterr()
A macro that copies the error structure out of the client handle to the structure at address *errp*.

clnt_pcreateerror()
Print a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string *s* and a colon. A newline character is appended at the end of the message. Used when a **clnt_create**(), **clntraw_create**(), **clnttcp_create**(), or **clntudp_create**() call fails.

clnt_perrno()
Print a message to standard error corresponding to the condition indicated by *stat*. A newline character is appended at the end of the message. Used after **callrpc**().

clnt_perror()
Print a message to standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. A newline character is appended at the end of the message. Used after **clnt_call**().

clnt_spcreateerror()
Like **clnt_pcreateerror**(), except that it returns a string instead of printing to the standard error.

Bugs: returns pointer to static data that is overwritten on each call.

clnt_sperrno()
Take the same arguments as **clnt_perrno**(), but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which contains the message.

clnt_sperrno() is used instead of **clnt_perrno**() if the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with printf(3), or if a message format different than that supported by

clnt_perrno() is to be used. Note: unlike **clnt_sperror**() and **clnt_spcreateerror**(), **clnt_sperrno**() returns a pointer to static data, but the result will not get overwritten on each call.

clnt_sperror()
Like **clnt_perror**(), except that (like **clnt_sperrno**()) it returns a string instead of printing to standard error.

Bugs: returns pointer to static data that is overwritten on each call.

clntraw_create()
This routine creates a toy RPC client for the remote program *prognum*, version *versnum*. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see **svcraw_create**(). This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

clnttcp_create()
This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses TCP/IP as a transport. The remote program is located at Internet address ∗*addr*. If *addr->sin_port* is zero, then it is set to the actual port that the remote program is listening on (the remote rpcbind(8) or **portmap** service is consulted for this information). The parameter *sockp* is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp*. Since TCP-based RPC uses buffered I/O , the user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of zero choose suitable defaults. This routine returns NULL if it fails.

clntudp_create()
This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses UDP/IP as a transport. The remote program is located at Internet address *addr*. If *addr->sin_port* is zero, then it is set to actual port that the remote program is listening on (the remote rpcbind(8) or **portmap** service is consulted for this information). The parameter *sockp* is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp*. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by *clnt_call*.

Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

clntudp_bufcreate()
This routine creates an RPC client for the remote program *prognum*, on *versnum*; the client uses UDP/IP as a transport. The remote program is located at Internet address *addr*. If *addr->sin_port* is zero, then it is set to actual port that the remote program is listening on (the remote rpcbind(8) or **portmap** service is consulted for this information). The parameter *sockp* is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp*. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by *clnt_call*.

This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

get_myaddress()
Stuff the machine's IP address into ∗*addr*, without consulting the library routines that deal with /etc/hosts. The port number is always set to **htons**(*PMAPPORT*). Returns zero on success, nonzero on failure.

pmap_getmaps()
A user interface to the rpcbind(8) service, which returns a list of the current RPC program-to-port mappings on the host located at IP address **addr*. This routine can return NULL. The command

rpcinfo -p

uses this routine.

pmap_getport()
A user interface to the rpcbind(8) service, which returns the port number on which waits a service that supports program number *prognum*, version *versnum*, and speaks the transport protocol associated with *protocol*. The value of *protocol* is most likely IPPROTO_UDP or IPPROTO_TCP. A return value of zero means that the mapping does not exist or that the RPC system failured to contact the remote rpcbind(8) service. In the latter case, the global variable **rpc_createerr**() contains the RPC status.

pmap_rmtcall()
A user interface to the rpcbind(8) service, which instructs rpcbind(8) on the host at IP address **addr* to make an RPC call on your behalf to a procedure on that host. The parameter **portp* will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in **callrpc**() and **clnt_call**(). This procedure should be used for a "ping" and nothing else. See also **clnt_broadcast**().

pmap_set()
A user interface to the rpcbind(8) service, which establishes a mapping between the triple [*prognum*, *versnum*, *protocol*] and *port* on the machine's rpcbind(8) service. The value of *protocol* is most likely IPPROTO_UDP or IPPROTO_TCP. This routine returns one if it succeeds, zero otherwise. Automatically done by **svc_register**().

pmap_unset()
A user interface to the rpcbind(8) service, which destroys all mapping between the triple [*prognum*, *versnum*, ***] and *ports* on the machine's rpcbind(8) service. This routine returns one if it succeeds, zero otherwise.

registerrpc()
Register procedure *procname* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter(s); *progname* should return a pointer to its static result(s); *inproc* is used to decode the parameters while *outproc* is used to encode the results. This routine returns zero if the registration succeeded, −1 otherwise.

Warning: remote procedures registered in this form are accessed using the UDP/IP transport; see **svcudp_bufcreate**() for restrictions.

struct rpc_createerr rpc_createerr;
A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine **clnt_pcreateerror**() to print the reason why.

svc_destroy()
A macro that destroys the RPC service transport handle, *xprt*. Destruction usually involves deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after calling this routine.

fd_set svc_fdset;
A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the select(2) system call. This is only of interest if a service implementor does not call **svc_run**(), but rather does his own asynchronous event processing. This variable is read-only (do not pass its address to select(2)!), yet it may change after calls to **svc_getreqset**() or any creation

routines.

int svc_fds;
Similar to **svc_fedset**(), but limited to 32 descriptors. This interface is obsoleted by **svc_fdset**().

svc_freeargs()
A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **svc_getargs**(). This routine returns 1 if the results were successfully freed, and zero otherwise.

svc_getargs()
A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, *xprt*. The parameter *in* is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns one if decoding succeeds, and zero otherwise.

svc_getcaller()
The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle, *xprt*.

svc_getreqset()
This routine is only of interest if a service implementor does not call **svc_run**(), but instead implements custom asynchronous event processing. It is called when the select(2) system call has determined that an RPC request has arrived on some RPC socket(s) ; *rdfds* is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfds* have been serviced.

svc_getreq()
Similar to **svc_getreqset**(), but limited to 32 descriptors. This interface is obsoleted by **svc_getreqset**().

svc_register()
Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *protocol* is zero, the service is not registered with the rpcbind(8) service. If *protocol* is non-zero, then a mapping of the triple [*prognum*, *versnum*, *protocol*] to *xprt->xp_port* is established with the local rpcbind(8) service (generally *protocol* is zero, IPPROTO_UDP or IPPROTO_TCP). The procedure *dispatch* has the following form: *int* **dispatch**(*struct svc_req* ∗*request*, *SVCXPRT* ∗*xprt*).

The **svc_register**() routine returns one if it succeeds, and zero otherwise.

svc_run()
This routine never returns. It waits for RPC requests to arrive, and calls the appropriate service procedure using **svc_getreq**() when one arrives. This procedure is usually waiting for a select(2) system call to return.

svc_sendreply()
Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter *xprt* is the request's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns one if it succeeds, zero otherwise.

svc_unregister()
Remove all mapping of the double [*prognum*, *versnum*] to dispatch routines, and of the triple [*prognum*, *versnum*, ∗] to port number.

svcerr_auth()
Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

svcerr_decode()
Called by a service dispatch routine that cannot successfully decode its parameters. See also **svc_getargs**().

svcerr_noproc()
Called by a service dispatch routine that does not implement the procedure number that the caller requests.

svcerr_noprog()
Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

svcerr_progvers()
Called when the desired version of a program is not registered with the RPC package. Service implementors usually do not need this routine.

svcerr_systemerr()
Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

svcerr_weakauth()
Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient authentication parameters. The routine calls **svcerr_auth**(*xprt*, *AUTH_TOOWEAK*).

svcraw_create()
This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see **clntraw_create**(). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

svctcp_create()
This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, *xprt->xp_sock* is the transport's socket descriptor, and *xprt->xp_port* is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O , users may specify the size of buffers; values of zero choose suitable defaults.

svcfd_create()
Create a service on top of any open descriptor. Typically, this descriptor is a connected socket for a stream protocol such as TCP. *sendsize* and *recvsize* indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

svcudp_bufcreate()
This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, *xprt->xp_sock* is the transport's socket descriptor, and *xprt->xp_port* is the transport's port number. This routine returns NULL if it fails.

This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

svcudp_create()
This acts as **svcudp_bufcreate**(*with*) predefined sizes for the maximum packet sizes.

xdr_accepted_reply()
Used for encoding RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_authunix_parms()
Used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

xdr_callhdr()
Used for describing RPC call header messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_callmsg()
Used for describing RPC call messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_opaque_auth()
Used for describing RPC authentication information messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_pmap()
Used for describing parameters to various rpcbind(8) procedures, externally. This routine is useful for users who wish to generate these parameters without using the *pmap* interface.

xdr_pmaplist()
Used for describing a list of port mappings, externally. This routine is useful for users who wish to generate these parameters without using the *pmap* interface.

xdr_rejected_reply()
Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_replymsg()
Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xprt_register()
After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable *svc_fds*. Service implementors usually do not need this routine.

xprt_unregister()
Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable *svc_fds*. Service implementors usually do not need this routine.

SEE ALSO

xdr(3)

The following manuals:

Remote Procedure Calls: Protocol Specification.

Remote Procedure Call Programming Guide.

rpcgen Programming Guide.

Sun Microsystems, Inc., USC-ISI, "RPC: Remote Procedure Call Protocol Specification", *RFC*, 1050.

NAME

svc_dg_enablecache, **svc_exit**, **svc_fdset**, **svc_freeargs**, **svc_getargs**, **svc_getreq_common**, **svc_getreq_poll**, **svc_getreqset**, **svc_getrpccaller**, **svc_pollset**, **svc_run**, **svc_sendreply** — library routines for RPC servers

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <rpc/rpc.h>

int
svc_dg_enablecache(SVCXPRT *xprt, const unsigned cache_size);

void
svc_exit(void);

bool_t
svc_freeargs(const SVCXPRT *xprt, const xdrproc_t inproc, caddr_t in);

bool_t
svc_getargs(const SVCXPRT *xprt, const xdrproc_t inproc, caddr_t in);

void
svc_getreq_common(const int fd);

void
svc_getreq_poll(struct pollfd *pfdp, const int pollretval);

void
svc_getreqset(fd_set * rdfds);

struct netbuf *
svc_getrpccaller(const SVCXPRT *xprt);

struct sockcred *
__svc_getcallercreds(const SVCXPRT *xprt);

struct pollfd svc_pollset[FD_SETSIZE];

void
svc_run(void);

bool_t
svc_sendreply(const SVCXPRT *xprt, const xdrproc_t outproc,
    const caddr_t *out);
```

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network.

These routines are associated with the server side of the RPC mechanism. Some of them are called by the server side dispatch function, while others (such as **svc_run**()) are called when the server is initiated.

ROUTINES

See rpc(3) for the definition of the SVCXPRT data structure.

svc_dg_enablecache() This function allocates a duplicate request cache for the service endpoint *xprt*, large enough to hold *cache_size* entries. Once enabled, there is no way to disable caching. This routine returns 0 if space necessary for a cache of the given size was successfully allocated, and 1 otherwise.

svc_exit() This function when called by any of the RPC server procedure or otherwise, causes **svc_run**() to return.

As currently implemented, **svc_exit**() zeroes the svc_fdset global variable. If RPC server activity is to be resumed, services must be reregistered with the RPC library either through one of the **rpc_svc_create**() functions, or using **xprt_register**(). **svc_exit**() has global scope and ends all RPC server activity. *fd_set svc_fdset* A global variable reflecting the RPC server's read file descriptor bit mask; it is suitable as a parameter to the select(2) system call. This is only of interest if service implementors do not call **svc_run**(), but rather do their own asynchronous event processing. This variable is read-only (do not pass its address to select(2)!), yet it may change after calls to **svc_getreqset**() or any creation routines.

svc_freeargs() A function macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **svc_getargs**(). This routine returns TRUE if the results were successfully freed, and FALSE otherwise.

svc_getargs() A function macro that decodes the arguments of an RPC request associated with the RPC service transport handle *xprt*. The parameter *in* is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns TRUE if decoding succeeds, and FALSE otherwise.

svc_getreq_common() This routine is called to handle a request on the given file descriptor.

svc_getreq_poll() This routine is only of interest if a service implementor does not call **svc_run**(), but instead implements custom asynchronous event processing. It is called when poll(2) has determined that an RPC request has arrived on some RPC file descriptors; **pollretval**() is the return value from poll(2) and *pfdp* is the array of *pollfd* structures on which the poll(2) was done. It is assumed to be an array large enough to contain the maximal number of descriptors allowed.

svc_getreqset() This routine is only of interest if a service implementor does not call **svc_run**(), but instead implements custom asynchronous event processing. It is called when poll(2) has determined that an RPC request has arrived on some RPC file descriptors; *rdfds* is the resultant read file descriptor bit mask. The routine returns when all file descriptors associated with the value of *rdfds* have been serviced.

svc_getrpccaller() The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle *xprt*.

__svc_getcallercreds() *Warning: this macro is specific to* NetBSD *and thus not portable.* This macro returns a pointer to a sockcred structure, defined in ⟨sys/socket.h⟩, identifying the calling client. This only works if the client is calling the server over an AF_LOCAL socket.

struct pollfd svc_pollset[FD_SETSIZE];
: *svc_pollset* is an array of *pollfd* structures derived from *svc_fdset[]*. It is suitable as a parameter to the poll(2) system call. The derivation of *svc_pollset* from *svc_fdset* is made in the current implementation in **svc_run**(). Service implementors who do not call **svc_run**() and who wish to use this array must perform this derivation themselves.

svc_run()
: This routine never returns. It waits for RPC requests to arrive, and calls the appropriate service procedure using **svc_getreq_poll**() when one arrives. This procedure is usually waiting for the poll(2) system call to return.

svc_sendreply()
: Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter *xprt* is the request's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns TRUE if it succeeds, FALSE otherwise.

SEE ALSO

poll(2), rpc(3), rpc_svc_create(3), rpc_svc_err(3), rpc_svc_reg(3)

NAME

rpc_svc_create, **svc_control**, **svc_create**, **svc_destroy**, **svc_dg_create**, **svc_fd_create**, **svc_raw_create**, **svc_tli_create**, **svc_tp_create**, **svc_vc_create** — library routines for the creation of server handles

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <rpc/rpc.h>

bool_t
svc_control(SVCXPRT *svc, const u_int req, void *info);

int
svc_create(const void (*dispatch)(struct svc_req *, SVCXPRT *),
    const rpcprog_t prognum, const rpcvers_t versnum, const char *nettype);

SVCXPRT *"
svc_dg_create(const int fildes, const u_int sendsz, const u_int recvsz);

void
svc_destroy(SVCXPRT *xprt);

SVCXPRT *
svc_fd_create(const int fildes, const u_int sendsz, const u_int recvsz);

SVCXPRT *
svc_raw_create(void);

SVCXPRT *
svc_tli_create(const int fildes, const struct netconfig *netconf,
    const struct t_bind *bindaddr, const u_int sendsz, const u_int recvsz);

SVCXPRT *
svc_tp_create(const void (*dispatch)(const struct svc_reg *, const SVCXPRT *),
    const rpcprog_t prognum, const rpcvers_t versnum,
    const struct netconfig *netconf);

SVCXPRT *
svc_vc_create(const int fildes, const u_int sendsz, const u_int recvsz);
```

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on servers across the network. These routines deal with the creation of service handles. Once the handle is created, the server can be invoked by calling **svc_run**().

ROUTINES

See rpc(3) for the definition of the SVCXPRT data structure.

svc_control()

A function to change or retrieve various information about a service object. *req* indicates the type of operation and *info* is a pointer to the information. The supported values of *req*, their argument types, and what they do are:

SVCGET_VERSQUIET
: If a request is received for a program number served by this server but the version number is outside the range registered with the server, an RPC_PROGVERSMISMATCH error will normally be returned. *info* should be a pointer to an integer. Upon successful completion of the SVCGET_VERSQUIET request, **info* contains an integer which describes the server's current behavior: 0 indicates normal server behavior (that is, an RPC_PROGVERSMISMATCH error will be returned); 1 indicates that the out of range request will be silently ignored.

SVCSET_VERSQUIET
: If a request is received for a program number served by this server but the version number is outside the range registered with the server, an RPC_PROGVERSMISMATCH error will normally be returned. It is sometimes desirable to change this behavior. *info* should be a pointer to an integer which is either 0 (indicating normal server behavior - an RPC_PROGVERSMISMATCH error will be returned), or 1 (indicating that the out of range request should be silently ignored).

svc_create()
: **svc_create**() creates server handles for all the transports belonging to the class *nettype*. *nettype* defines a class of transports which can be used for a particular application. The transports are tried in left to right order in NETPATH variable or in top to bottom order in the netconfig database. If *nettype* is NULL, it defaults to netpath.

  svc_create() registers itself with the rpcbind service (see rpcbind(8)). *dispatch* is called when there is a remote procedure call for the given *prognum* and *versnum*; this requires calling **svc_run**() (see **svc_run**() in rpc_svc_reg(3)). If **svc_create**() succeeds, it returns the number of server handles it created, otherwise it returns 0 and an error message is logged.

svc_destroy()
: A function macro that destroys the RPC service handle *xprt*. Destruction usually involves deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after calling this routine.

svc_dg_create()
: This routine creates a connectionless RPC service handle, and returns a pointer to it. This routine returns NULL if it fails, and an error message is logged. *sendsz* and *recvsz* are parameters used to specify the size of the buffers. If they are 0, suitable defaults are chosen. The file descriptor *fildes* should be open and bound. The server is not registered with rpcbind(8). Warning: since connectionless-based RPC messages can only hold limited amount of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

svc_fd_create()
: This routine creates a service on top of an open and bound file descriptor, and returns the handle to it. Typically, this descriptor is a connected file descriptor for a connection-oriented transport. *sendsz* and *recvsz* indicate sizes for the send and receive buffers. If they are 0, reasonable defaults are chosen. This routine returns NULL if it fails, and an error message is logged.

svc_raw_create()
: This routine creates an RPC service handle and returns a pointer to it. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; (see **clnt_raw_create**() in rpc_clnt_create(3)). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel and networking interference. This routine returns NULL if it fails, and an error message is logged. Note: **svc_run**() should not be called when the raw interface is being used.

svc_tli_create()

This routine creates an RPC server handle, and returns a pointer to it. *fildes* is the file descriptor on which the service is listening. If *fildes* is RPC_ANYFD, it opens a file descriptor on the transport specified by *netconf*. If the file descriptor is unbound and *bindaddr* is non-null *fildes* is bound to the address specified by *bindaddr*, otherwise *fildes* is bound to a default address chosen by the transport.

Note: the *t_bind* structure comes from the TLI/XTI SysV interface, which NetBSD does not use. The structure is defined in ⟨rpc/types.h⟩ for compatibility as:

```
struct t_bind {
    struct netbuf addr;       /* network address, see rpc(3) */
    unsigned int  qlen;       /* queue length (for listen(2)) */
};
```

In the case where the default address is chosen, the number of outstanding connect requests is set to 8 for connection-oriented transports. The user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. This routine returns NULL if it fails, and an error message is logged. The server is not registered with the rpcbind(8) service.

svc_tp_create()

svc_tp_create() creates a server handle for the network specified by *netconf*, and registers itself with the rpcbind service. *dispatch* is called when there is a remote procedure call for the given *prognum* and *versnum*; this requires calling **svc_run**(). **svc_tp_create**() returns the service handle if it succeeds, otherwise a NULL is returned and an error message is logged.

svc_vc_create()

This routine creates a connection-oriented RPC service and returns a pointer to it. This routine returns NULL if it fails, and an error message is logged. The users may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. The file descriptor *fildes* should be open and bound. The server is not registered with the rpcbind(8) service.

SEE ALSO

rpc(3), rpc_svc_calls(3), rpc_svc_err(3), rpc_svc_reg(3), rpcbind(8)

NAME

rpc_svc_err, **svcerr_auth**, **svcerr_decode**, **svcerr_noproc**, **svcerr_noprog**, **svcerr_progvers**, **svcerr_systemerr**, **svcerr_weakauth** — library routines for server side remote procedure call errors

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <rpc/rpc.h>

void
svcerr_auth(const SVCXPRT1 *xprt, const enum auth_stat why);

void
svcerr_decode(const SVCXPRT *xprt);

void
svcerr_noproc(const SVCXPRT *xprt);

void
svcerr_noprog(const SVCXPRT *xprt);

void
svcerr_progvers(const SVCXPRT *xprt, rpcvers_t low_vers,
    rpcvers_t high_vers);

void
svcerr_systemerr(const SVCXPRT *xprt);

void
svcerr_weakauth(const SVCXPRT *xprt);
```

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network.

These routines can be called by the server side dispatch function if there is any error in the transaction with the client.

ROUTINES

See rpc(3) for the definition of the *SVCXPRT* data structure.

svcerr_auth()

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

svcerr_decode() Called by a service dispatch routine that cannot successfully decode the remote parameters (see **svc_getargs**() in rpc_svc_reg(3)).

svcerr_noproc()

Called by a service dispatch routine that does not implement the procedure number that the caller requests.

svcerr_noprog()

Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

svcerr_progvers()
Called when the desired version of a program is not registered with the RPC package. *low_vers* is the lowest version number, and *high_vers* is the highest version number. Service implementors usually do not need this routine.

svcerr_systemerr()
Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

svcerr_weakauth()
Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication parameters. The routine calls **svcerr_auth**(*xprt*, *AUTH_TOOWEAK*).

SEE ALSO

rpc(3), rpc_svc_calls(3), rpc_svc_create(3), rpc_svc_reg(3)

NAME

rpc_svc_reg, **rpc_reg**, **svc_reg**, **svc_unreg**, **svc_auth_reg**, **xprt_register**, **xprt_unregister** — library routines for registering servers

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <rpc/rpc.h>

bool_t
rpc_reg(const rpcprog_t prognum, const rpcvers_t versnum,
    const rpcproc_t procnum, const char *(*procname)(),
    const xdrproc_t inproc, const xdrproc_t outproc, const char *nettype);

int
svc_reg(const SVCXPRT *xprt, const rpcprog_t prognum,
    const rpcvers_t versnum,
    const void (*dispatch(struct svc_req *, SVCXPRT *),
    const struct netconfig *netconf);

void
svc_unreg(const rpcprog_t prognum, const rpcvers_t versnum);

int
svc_auth_reg(const int cred_flavor,
    const enum auth_stat (*handler(struct svc_req *, struct rpc_msg *)));

void
xprt_register(const SVCXPRT *xprt);

void
xprt_unregister(const SVCXPRT *xprt);
```

DESCRIPTION

These routines are a part of the RPC library which allows the RPC servers to register themselves with rpcbind (see rpcbind(8)), and associate the given program and version number with the dispatch function. When the RPC server receives a RPC request, the library invokes the dispatch routine with the appropriate arguments.

ROUTINES

See rpc(3) for the definition of the *SVCXPRT* data structure.

rpc_reg()

Register program *prognum*, procedure *procname*, and version *versnum* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter(s); *procname* should return a pointer to its static result(s); *inproc* is the XDR function used to decode the parameters while *outproc* is the XDR function used to encode the results. Procedures are registered on all available transports of the class *nettype*. See rpc(3). This routine returns 0 if the registration succeeded, -1 otherwise.

svc_reg()

Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *netconf* is NULL, the service is not registered with the rpcbind(8) service. If *netconf* is non-zero, then a mapping of the triple [*prognum*, *versnum*, *netconf->nc_netid*] to *xprt->xp_ltaddr* is established with the local rpcbind service.

The **svc_reg**() routine returns 1 if it succeeds, and 0 otherwise.

svc_unreg()
Remove from the rpcbind service, all mappings of the triple [*prognum*, *versnum*, all-transports] to network address and all mappings within the RPC service package of the double [*prognum*, *versnum*] to dispatch routines.

svc_auth_reg()
Registers the service authentication routine *handler* with the dispatch mechanism so that it can be invoked to authenticate RPC requests received with authentication type *cred_flavor*. This interface allows developers to add new authentication types to their RPC applications without needing to modify the libraries. Service implementors usually do not need this routine.

Typical service application would call **svc_auth_reg**() after registering the service and prior to calling **svc_run**(). When needed to process an RPC credential of type *cred_flavor*, the *handler* procedure will be called with two parameters *struct svc_req *rqst*, and *struct rpc_msg * msg*, and is expected to return a valid *enum auth_stat* value. There is no provision to change or delete an authentication handler once registered.

The **svc_auth_reg**() routine returns 0 if the registration is successful, 1 if *cred_flavor* already has an authentication handler registered for it, and -1 otherwise.

xprt_register()
After RPC service transport handle *xprt* is created, it is registered with the RPC service package. This routine modifies the global variable *svc_fdset* (see rpc_svc_calls(3)). Service implementors usually do not need this routine.

xprt_unregister()
Before an RPC service transport handle *xprt* is destroyed, it unregisters itself with the RPC service package. This routine modifies the global variable *svc_fdset* (see rpc_svc_calls(3)). Service implementors usually do not need this routine.

SEE ALSO

select(2), rpc(3), rpc_svc_calls(3), rpc_svc_create(3), rpc_svc_err(3), rpcbind(3), rpcbind(8)

NAME

xdr_accepted_reply, **xdr_authsys_parms**, **xdr_callhdr**, **xdr_callmsg**, **xdr_opaque_auth**, **xdr_rejected_reply**, **xdr_replymsg** — XDR library routines for remote procedure calls

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <rpc/rpc.h>

bool_t
xdr_accepted_reply(XDR *xdrs, const struct accepted_reply *ar);

bool_t
xdr_authsys_parms(XDR *xdrs, struct authsys_parms *aupp);

void
xdr_callhdr(XDR *xdrs, struct rpc_msg *chdr);

bool_t
xdr_callmsg(XDR *xdrs, struct rpc_msg *cmsg);

bool_t
xdr_opaque_auth(XDR *xdrs, struct opaque_auth *ap);

bool_t
xdr_rejected_reply(XDR *xdrs, const struct rejected_reply *rr);

bool_t
xdr_replymsg(XDR *xdrs, const struct rpc_msg *rmsg);
```

DESCRIPTION

These routines are used for describing the RPC messages in XDR language. They should normally be used by those who do not want to use the RPC package directly. These routines return TRUE if they succeed, FALSE otherwise.

ROUTINES

See rpc(3) for the definition of the *XDR* data structure.

xdr_accepted_reply()
: Used to translate between RPC reply messages and their external representation. It includes the status of the RPC call in the XDR language format. In the case of success, it also includes the call results.

xdr_authsys_parms()
: Used for describing UNIX operating system credentials. It includes machine-name, uid, gid list, etc.

xdr_callhdr()
: Used for describing call header messages. It encodes the static part of the call message header in the XDR language format. It includes information such as transaction ID, RPC version number, program and version number.

xdr_callmsg()
: Used for describing RPC call messages. This includes all the RPC call information such as transaction ID, RPC version number, program number, version number, authentication information, etc. This is normally used by servers to determine information about the client RPC call.

xdr_opaque_auth()
Used for describing RPC opaque authentication information messages.

xdr_rejected_reply()
Used for describing RPC reply messages. It encodes the rejected RPC message in the XDR language format. The message could be rejected either because of version number mis-match or because of authentication errors.

xdr_replymsg()
Used for describing RPC reply messages. It translates between the RPC reply message and its external representation. This reply could be either an acceptance, rejection or NULL.

SEE ALSO

rpc(3), xdr(3)

NAME

rpcb_getmaps, **rpcb_getaddr**, **rpcb_gettime**, **rpcb_rmtcall**, **rpcb_set**, **rpcb_unset** — library routines for RPC bind service

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <rpc/rpc.h>

struct rpcblist *
rpcb_getmaps(const struct netconfig *netconf, const char *host);

bool_t
rpcb_getaddr(const rpcprog_t prognum, const rpcvers_t versnum,
    const struct netconfig *netconf, struct netbuf *svcaddr,
    const char *host);

bool_t
rpcb_gettime(const char *host, time_t * timep);

enum clnt_stat
rpcb_rmtcall(const struct netconfig *netconf, const char *host,
    const rpcprog_t prognum, const rpcvers_t versnum,
    const rpcproc_t procnum, const xdrproc_t inproc, const char *in,
    const xdrproc_t outproc, caddr_t out,
    const struct timeval tout, struct netbuf *svcaddr);

bool_t
rpcb_set(const rpcprog_t prognum, const rpcvers_t versnum,
    const struct netconfig *netconf, const struct netbuf *svcaddr);

bool_t
rpcb_unset(const rpcprog_t prognum, const rpcvers_t versnum,
    const struct netconfig *netconf);
```

DESCRIPTION

These routines allow client C programs to make procedure calls to the RPC binder service. (see `rpcbind`(8)) maintains a list of mappings between programs and their universal addresses.

ROUTINES

rpcb_getmaps()

An interface to the rpcbind service, which returns a list of the current RPC program-to-address mappings on *host*. It uses the transport specified through *netconf* to contact the remote rpcbind service on *host*. This routine will return `NULL`, if the remote rpcbind could not be contacted.

rpcb_getaddr()

An interface to the rpcbind service, which finds the address of the service on *host* that is registered with program number *prognum*, version *versnum*, and speaks the transport protocol associated with *netconf*. The address found is returned in *svcaddr*. *svcaddr* should be preallocated. This routine returns `TRUE` if it succeeds. A return value of `FALSE` means that the mapping does not exist or that the RPC system failed to contact the remote rpcbind service. In the latter case, the global variable *rpc_createerr* (see `rpc_clnt_create`(3) contains the RPC status.

rpcb_gettime()
This routine returns the time on *host* in *timep*. If *host* is NULL, **rpcb_gettime**() returns the time on its own machine. This routine returns TRUE if it succeeds, FALSE if it fails. **rpcb_gettime**() can be used to synchronize the time between the client and the remote server.

rpcb_rmtcall()
An interface to the rpcbind service, which instructs rpcbind on *host* to make an RPC call on your behalf to a procedure on that host. The **netconfig**() structure should correspond to a connectionless transport. The parameter *svcaddr* will be modified to the server's address if the procedure succeeds (see **rpc_call**() and **clnt_call**() in rpc_clnt_calls(3) for the definitions of other parameters).

This procedure should normally be used for a ''ping'' and nothing else. This routine allows programs to do lookup and call, all in one step. Note: Even if the server is not running **rpcb_rmtcall**() does not return any error messages to the caller. In such a case, the caller times out.

Note: **rpcb_rmtcall**() is only available for connectionless transports.

rpcb_set()
An interface to the rpcbind service, which establishes a mapping between the triple [*prognum*, *versnum*, *netconf->nc_netid*] and *svcaddr* on the machine's rpcbind service. The value of *nc_netid* must correspond to a network identifier that is defined by the netconfig database. This routine returns TRUE if it succeeds, FALSE otherwise. (See also in rpc_svc_calls(3). If there already exists such an entry with rpcbind, **rpcb_set**() will fail.

rpcb_unset()
An interface to the rpcbind service, which destroys the mapping between the triple [*prognum*, *versnum*, *netconf->nc_netid*] and the address on the machine's rpcbind service. If *netconf* is NULL, **rpcb_unset**() destroys all mapping between the triple [*prognum*, *versnum*, *all-transports*] and the addresses on the machine's rpcbind service. This routine returns TRUE if it succeeds, FALSE otherwise. Only the owner of the service or the super-user can destroy the mapping. (See also **svc_unreg**() in rpc_svc_calls(3).

SEE ALSO
rpc_clnt_calls(3), rpc_svc_calls(3), rpcbind(8), rpcinfo(8)

NAME

rtbl_create, **rtbl_destroy**, **rtbl_set_flags**, **rtbl_get_flags**, **rtbl_set_prefix**, **rtbl_set_separator**, **rtbl_set_column_prefix**, **rtbl_set_column_affix_by_id**, **rtbl_add_column**, **rtbl_add_column_by_id**, **rtbl_add_column_entry**, **rtbl_add_column_entry_by_id**, **rtbl_new_row**, **rtbl_format** — format data in simple tables

LIBRARY

The roken library (libroken, -lroken)

SYNOPSIS

```
#include <rtbl.h>

int
rtbl_add_column(rtbl_t table, const char *column_name, unsigned int flags);

int
rtbl_add_column_by_id(rtbl_t table, unsigned int column_id,
    const char *column_header, unsigned int flags);

int
rtbl_add_column_entry(rtbl_t table, const char *column_name,
    const char *cell_entry);

int
rtbl_add_column_entry_by_id(rtbl_t table, unsigned int column_id,
    const char *cell_entry);

rtbl_t
rtbl_create(void);

void
rtbl_destroy(rtbl_t table);

int
rtbl_new_row(rtbl_t table);

int
rtbl_set_column_affix_by_id(rtbl_t table, unsigned int column_id, const,
    char, *prefix", const char *suffix);

int
rtbl_set_column_prefix(rtbl_t table, const char *column_name,
    const char *prefix);

unsigned int
rtbl_get_flags(rtbl_t table);

void
rtbl_set_flags(rtbl_t table, unsigned int flags);

int
rtbl_set_prefix(rtbl_t table, const char *prefix);

int
rtbl_set_separator(rtbl_t table, const char *separator);

int
rtbl_format(rtbl_t table, FILE, *file");
```

DESCRIPTION

This set of functions assemble a simple table consisting of rows and columns, allowing it to be printed with certain options. Typical use would be output from tools such as `ls`(1) or `netstat`(1), where you have a fixed number of columns, but don't know the column widthds before hand.

A table is created with **`rtbl_create`**() and destroyed with **`rtbl_destroy`**().

Global flags on the table are set with *`rtbl_set_flags`* and retrieved with *`rtbl_get_flags`*. At present the only defined flag is `RTBL_HEADER_STYLE_NONE` which suppresses printing the header.

Before adding data to the table, one or more columns need to be created. This would normally be done with **`rtbl_add_column_by_id`**(), *`column_id`* is any number of your choice (it's used only to identify columns), *`column_header`* is the header to print at the top of the column, and *`flags`* are flags specific to this column. Currently the only defined flag is `RTBL_ALIGN_RIGHT`, aligning column entries to the right. Columns are printed in the order they are added.

There's also a way to add columns by column name with **`rtbl_add_column`**(), but this is less flexible (you need unique header names), and is considered deprecated.

To add data to a column you use **`rtbl_add_column_entry_by_id`**(), where the *`column_id`* is the same as when the column was added (adding data to a non-existent column is undefined), and *`cell_entry`* is whatever string you wish to include in that cell. It should not include newlines. For columns added with **`rtbl_add_column`**() you must use **`rtbl_add_column_entry`**() instead.

`rtbl_new_row`() fills all columns with blank entries until they all have the same number of rows.

Each column can have a separate prefix and suffix, set with *`rtbl_set_column_affix_by_id`*; *`rtbl_set_column_prefix`* allows setting the prefix only by column name. In addition to this, columns may be separated by a string set with *`rtbl_set_separator`* (by default columns are not seprated by anything).

The finished table is printed to *`file`* with *`rtbl_format`*.

EXAMPLES

This program:

```
#include <stdio.h>
#include <rtbl.h>
int
main(int argc, char **argv)
{
    rtbl_t table;
    table = rtbl_create();
    rtbl_set_separator(table, "  ");
    rtbl_add_column_by_id(table, 0, "Column A", 0);
    rtbl_add_column_by_id(table, 1, "Column B", RTBL_ALIGN_RIGHT);
    rtbl_add_column_by_id(table, 2, "Column C", 0);
    rtbl_add_column_entry_by_id(table, 0, "A-1");
    rtbl_add_column_entry_by_id(table, 0, "A-2");
    rtbl_add_column_entry_by_id(table, 0, "A-3");
    rtbl_add_column_entry_by_id(table, 1, "B-1");
    rtbl_add_column_entry_by_id(table, 2, "C-1");
    rtbl_add_column_entry_by_id(table, 2, "C-2");
    rtbl_add_column_entry_by_id(table, 1, "B-2");
    rtbl_add_column_entry_by_id(table, 1, "B-3");
    rtbl_add_column_entry_by_id(table, 2, "C-3");
```

```
        rtbl_add_column_entry_by_id(table, 0, "A-4");
        rtbl_new_row(table);
        rtbl_add_column_entry_by_id(table, 1, "B-4");
        rtbl_new_row(table);
        rtbl_add_column_entry_by_id(table, 2, "C-4");
        rtbl_new_row(table);
        rtbl_format(table, stdout);
        rtbl_destroy(table);
        return 0;
    }
```

will output the following:

```
    Column A  Column B  Column C
    A-1            B-1  C-1
    A-2            B-2  C-2
    A-3            B-3  C-3
    A-4
                   B-4
                        C-4
```

NAME

scandir, **alphasort** — scan a directory

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

int
scandir(const char *dirname, struct dirent ***namelist,
    int (*select)(const struct dirent *),
    int (*compar)(const void *, const void *));

int
alphasort(const void *d1, const void *d2);
```

DESCRIPTION

The **scandir**() function reads the directory *dirname* and builds an array of pointers to directory entries using malloc(3). It returns the number of entries in the array. A pointer to the array of directory entries is stored in the location referenced by *namelist*.

The *select* parameter is a pointer to a user supplied subroutine which is called by **scandir**() to select which entries are to be included in the array. The select routine is passed a pointer to a directory entry and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compar* parameter is a pointer to a user supplied subroutine which is passed to qsort(3) to sort the completed array. If this pointer is null, the array is not sorted.

The **alphasort**() function is a routine which can be used for the *compar* parameter to sort the array alphabetically.

The memory allocated for the array can be deallocated with free(3), by freeing each pointer in the array and then the array itself.

DIAGNOSTICS

Returns −1 if the directory cannot be opened for reading or if malloc(3) cannot allocate enough memory to hold all the data structures.

SEE ALSO

directory(3), malloc(3), qsort(3), dir(5)

HISTORY

The **scandir**() and **alphasort**() functions appeared in 4.2BSD.

NAME

scanf, **fscanf**, **sscanf**, **vscanf**, **vsscanf**, **vfscanf** — input format conversion

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

int
scanf(const char * restrict format, ...);

int
fscanf(FILE * restrict stream, const char * restrict format, ...);

int
sscanf(const char * restrict str, const char * restrict format, ...);

#include <stdarg.h>

int
vscanf(const char * restrict format, va_list ap);

int
vsscanf(const char * restrict str, const char * restrict format,
    va_list ap);

int
vfscanf(FILE * restrict stream, const char * restrict format, va_list ap);
```

DESCRIPTION

The **scanf**() family of functions scans input according to a *format* as described below. This format may contain *conversion specifiers*; the results from such conversions, if any, are stored through the *pointer* arguments. The **scanf**() function reads input from the standard input stream *stdin*, **fscanf**() reads input from the stream pointer *stream*, and **sscanf**() reads its input from the character string pointed to by *str*. The **vfscanf**() function is analogous to `vfprintf`(3) and reads input from the stream pointer *stream* using a variable argument list of pointers (see `stdarg`(3)). The **vscanf**() function scans a variable argument list from the standard input and the **vsscanf**() function scans it from a string; these are analogous to the **vprintf**() and **vsprintf**() functions respectively. Each successive *pointer* argument must correspond properly with each successive conversion specifier (but see 'suppression' below). All conversions are introduced by the **%** (percent sign) character. The *format* string may also contain other characters. White space (such as blanks, tabs, or newlines) in the *format* string match any amount of white space, including none, in the input. Everything else matches only itself. Scanning stops when an input character does not match such a format character. Scanning also stops when an input conversion cannot be made (see below).

CONVERSIONS

Following the **%** character introducing a conversion there may be a number of *flag* characters, as follows:

* Suppresses assignment. The conversion that follows occurs as usual, but no pointer is used; the result of the conversion is simply discarded.

h Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *short int* (rather than *int*).

hh Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *char* (rather than *int*).

j Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to an *intmax_t* (rather than *int*).

l Indicates either that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *long int* (rather than *int*), or that the conversion will be one of **efg** and the next pointer is a pointer to *double* (rather than *float*).

q Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *quad_t* (rather than *int*).

t Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *ptrdiff_t* (rather than *int*).

z Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *size_t* (rather than *int*).

L Indicates that the conversion will be **efg** and the next pointer is a pointer to *long double*.

In addition to these flags, there may be an optional maximum field width, expressed as a decimal integer, between the **%** and the conversion. If no width is given, a default of 'infinity' is used (with one exception, below); otherwise at most this many characters are scanned in processing the conversion. Before conversion begins, most conversions skip white space; this white space is not counted against the field width.

The following conversions are available:

% Matches a literal '%'. That is, '%%' in the format string matches a single input '%' character. No conversion is done, and assignment does not occur.

d Matches an optionally signed decimal integer; the next pointer must be a pointer to *int*.

D Equivalent to **ld**; this exists only for backwards compatibility.

i Matches an optionally signed integer; the next pointer must be a pointer to *int*. The integer is read in base 16 if it begins with '0x' or '0X', in base 8 if it begins with '0', and in base 10 otherwise. Only characters that correspond to the base are used.

o Matches an octal integer; the next pointer must be a pointer to *unsigned int*.

O Equivalent to **lo**; this exists for backwards compatibility.

u Matches an optionally signed decimal integer; the next pointer must be a pointer to *unsigned int*.

x Matches an optionally signed hexadecimal integer; the next pointer must be a pointer to *unsigned int*.

X Equivalent to **x**.

f Matches an optionally signed floating-point number; the next pointer must be a pointer to *float*.

e Equivalent to **f**.

g Equivalent to **f**.

E Equivalent to **f**.

G Equivalent to **f**.

s Matches a sequence of non-white-space characters; the next pointer must be a pointer to *char*, and the array must be large enough to accept all the sequence and the terminating NUL character. The input string stops at white space or at the maximum field width, whichever occurs first.

c Matches a sequence of *width* count characters (default 1); the next pointer must be a pointer to *char*, and there must be enough room for all the characters (no terminating NUL is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.

[Matches a nonempty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to *char*, and there must be enough room for all the characters in the string, plus a terminating NUL character. The usual skip of leading white space is suppressed. The string is to be made up of characters in (or not in) a particular set; the set is defined by the characters between the open bracket [character and a close bracket] character. The set *excludes* those characters if the first character after the open bracket is a circumflex ^. To include a close bracket in the set, make it the first character after the open bracket or the circumflex; any other position will end the set. The hyphen character **-** is also special; when placed between two other characters, it adds all intervening characters to the set. To include a hyphen, make it the last character before the final close bracket. For instance, [^]0-9-] means the set 'everything except close bracket, zero through nine, and hyphen'. The string ends with the appearance of a character not in the (or, with a circumflex, in) set or when the field width runs out.

p Matches a pointer value (as printed by '%p' in printf(3)); the next pointer must be a pointer to *void*.

n Nothing is expected; instead, the number of characters consumed thus far from the input is stored through the next pointer, which must be a pointer to *int*. This is *not* a conversion, although it can be suppressed with the * flag.

For backwards compatibility, other conversion characters (except '\0') are taken as if they were '%d' or, if uppercase, %ld, and a 'conversion' of %\0 causes an immediate return of EOF.

RETURN VALUES

These functions return the number of input items assigned, which can be fewer than provided for, or even zero, in the event of a matching failure. Zero indicates that, while there was input available, no conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a '%d' conversion. The value EOF is returned if an input failure occurs before any conversion such as an end-of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned.

SEE ALSO

getc(3), printf(3), strtod(3), strtol(3), strtoul(3)

STANDARDS

The functions **fscanf**(), **scanf**(), and **sscanf**() conform to ISO/IEC 9899:1990 ("ISO C90"). The **%j**, **%t** and **%z** conversion format modifiers conform to ISO/IEC 9899:1999 ("ISO C99"). The **vfscanf**(), **vscanf**() and **vsscanf**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

HISTORY

The functions **vscanf**(), **vsscanf**() and **vfscanf**() appeared in 4.4BSD or even 4.3BSD.

NOTES

All of the backwards compatibility formats will be removed in the future.

BUGS

Numerical strings are truncated to 512 characters; for example, **%f** and **%d** are implicitly **%512f** and **%512d**.

NAME

sched_setparam, **sched_getparam**, **sched_setscheduler**, **sched_getscheduler**, **sched_get_priority_max**, **sched_get_priority_min**, **sched_rr_get_interval**, **sched_yield** — process scheduling

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <sched.h>

int
sched_setparam(pid_t pid, const struct sched_param *param);

int
sched_getparam(pid_t pid, struct sched_param *param);

int
sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);

int
sched_getscheduler(pid_t pid);

int
sched_get_priority_max(int policy);

int
sched_get_priority_min(int policy);

int
sched_rr_get_interval(pid_t pid, struct timespec *interval);

int
sched_yield(void);
```

DESCRIPTION

This section describes the functions used to get the scheduling information about the processes, and control the scheduling of the process.

Available scheduling policies (classes) are:

`SCHED_OTHER` Time-sharing (TS) scheduling policy. The default policy in NetBSD.

`SCHED_FIFO` First in, first out (FIFO) scheduling policy.

`SCHED_RR` Round robin scheduling policy.

The *struct sched_param* contains at least one member:

sched_priority
: Specifies the priority of the process.

FUNCTIONS

sched_setparam(*pid*, *param*)
: Sets the scheduling parameters for the process specified by *pid* to *param*. If the value of *pid* is equal to zero, then the calling process is used.

sched_getparam(*pid*, *param*)
Gets the scheduling parameters of the process specified by *pid* into the structure *param*. If the value of *pid* is equal to zero, then the calling process is used.

sched_setscheduler(*pid*, *policy*, *param*)
Set the scheduling policy and parameters for the process specified by *pid*. If the value of *pid* is equal to zero, then the calling process is used.

sched_getscheduler(*pid*)
Returns the scheduling policy of the process specified by *pid*. If the value of *pid* is equal to zero, then the calling process is used.

sched_get_priority_max(*policy*)
Returns the maximal priority which may be used for the scheduling policy specified by *policy*.

sched_get_priority_min(*policy*)
Returns the minimal priority which may be used for the scheduling policy specified by *policy*.

sched_rr_get_interval(*pid*, *interval*)
Returns the time quantum into the structure *interval* of the process specified by *pid*. If the value of *pid* is equal to zero, then the calling process is used. The process must be running at *SCHED_RR* scheduling policy.

sched_yield()
Yields a processor voluntarily and gives other threads a chance to run without waiting for an involuntary preemptive switch.

RETURN VALUES

sched_setparam(), **sched_getparam**(), **sched_rr_get_interval**(), and **sched_yield**() return 0 on success. Otherwise, −1 is returned and *errno* is set to indicate the error.

sched_setscheduler() returns the previously used scheduling policy on success. Otherwise, −1 is returned and *errno* is set to indicate the error.

sched_getscheduler() returns the scheduling policy on success. Otherwise, −1 is returned and *errno* is set to indicate the error.

The **sched_get_priority_max**() and **sched_get_priority_min**() return the maximal/minimal priority value on success. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

The **sched_setparam**() and **sched_setscheduler**() functions fail if:

[EINVAL] At least one of the specified scheduling parameters was invalid.

[EPERM] The calling process has no appropriate privileges to perform the operation.

[ESRCH] No process can be found corresponding to that specified by *pid*, and value of *pid* is not zero.

The **sched_getparam**() and **sched_getscheduler**() functions fail if:

[EPERM] The calling process is not a super-user and its effective user id does not match the effective user-id of the specified process.

[ESRCH] No process can be found corresponding to that specified by *pid*, and value of *pid* is not zero.

The **sched_get_priority_max**() and **sched_get_priority_min**() functions fail if:

[EINVAL] The specified scheduling policy is invalid.

The **sched_rr_get_interval**() function fails if:

[ESRCH] No process can be found corresponding to that specified by *pid*, and value of *pid* is not zero.

SEE ALSO

pset(3), schedctl(8)

STANDARDS

These functions conform to IEEE Std 1003.1-2001 (“POSIX.1”) standard.

HISTORY

The scheduling functions appeared in NetBSD 5.0.

NAME

SDP_GET8, **SDP_GET16**, **SDP_GET32**, **SDP_GET64**, **SDP_GET128**, **SDP_GET_UUID128**, **SDP_PUT8**, **SDP_PUT16**, **SDP_PUT32**, **SDP_PUT64**, **SDP_PUT128**, **SDP_PUT_UUID128**, **sdp_open**, **sdp_open_local**, **sdp_close**, **sdp_error**, **sdp_search**, **sdp_attr2desc**, **sdp_uuid2desc** — Bluetooth SDP routines

LIBRARY

library “libsdp”

SYNOPSIS

```
#include <bluetooth.h>
#include <sdp.h>

SDP_GET8(b, cp);

SDP_GET16(s, cp);

SDP_GET32(l, cp);

SDP_GET64(l, cp);

SDP_GET128(l, cp);

SDP_GET_UUID128(l, cp);

SDP_PUT8(b, cp);

SDP_PUT16(s, cp);

SDP_PUT32(l, cp);

SDP_PUT64(l, cp);

SDP_PUT128(l, cp);

SDP_PUT_UUID128(l, cp);

void *
sdp_open(bdaddr_t const *l, bdaddr_t const *r);

void *
sdp_open_local(char const *control);

int32_t
sdp_close(void *xs);

int32_t
sdp_error(void *xs);

int32_t
sdp_search(void *xs, uint32_t plen, uint16_t const *pp, uint32_t alen,
    uint32_t const *ap, uint32_t vlen, sdp_attr_t *vp);

char const * const
sdp_attr2desc(uint16_t attr);

char const * const
sdp_uuid2desc(uint16_t uuid);

int32_t
sdp_register_service(void *xss, uint16_t uuid, bdaddr_t const *bdaddr,
    uint8_t const *data, uint32_t datalen, uint32_t *handle);
```

int32_t
sdp_unregister_service(*void *xss*, *uint32_t handle*);

int32_t
sdp_change_service(*void *xss*,*uint32_t handle*,*uint8_t const *data*,
uint32_t datalen);

DESCRIPTION

The **SDP_GET8**(), **SDP_GET16**(), **SDP_GET32**(), **SDP_GET64**() and **SDP_GET128**() macros are used to get byte, short, long, long long and 128-bit integer from the buffer pointed by *cp* pointer. The pointer is automatically advanced.

The **SDP_PUT8**(), **SDP_PUT16**(), **SDP_PUT32**(), **SDP_PUT64**() and **SDP_PUT128**() macros are used to put byte, short, long, long long and 128-bit integer into the buffer pointed by *cp* pointer. The pointer is automatically advanced.

SDP_GET_UUID128() and **SDP_PUT_UUID128**() macros are used to get and put 128-bit UUID into the buffer pointed by *cp* pointer. The pointer is automatically advanced.

The **sdp_open**() and **sdp_open_local**() functions each return a pointer to an opaque object describing SDP session. The *l* argument passed to **sdp_open**() function should point to a source BD_ADDR. If source BD_ADDR is `NULL` then source address `BDADDR_ANY` is used. The *r* argument passed to **sdp_open**() function should point to a non-`NULL` remote BD_ADDR. Remote BD_ADDR cannot be `BDADDR_ANY`. The **sdp_open_local**() function takes path to the control socket and opens a connection to a local SDP server. If path to the control socket is `NULL` then default `/var/run/sdp` path will be used.

The **sdp_close**() function terminates active SDP session and deletes SDP session object. The *xs* parameter should point to a valid SDP session object created with **sdp_open**() or **sdp_open_local**().

The **sdp_error**() function returns last error that is stored inside SDP session object. The *xs* parameter should point to a valid SDP session object created with **sdp_open**() or **sdp_open_local**(). The error value returned can be converted to a human readable message by calling `strerror`(3) function.

The **sdp_search**() function is used to perform SDP Service Search Attribute Request. The *xs* parameter should point to a valid SDP session object created with **sdp_open**() or **sdp_open_local**(). The *pp* parameter is a Service Search Pattern - an array of one or more Service Class IDs. The maximum number of Service Class IDs in the array is 12. The *plen* parameter is the length of the Service Search pattern. The *ap* parameter is an Attribute ID Range List - an array of one or more SDP Attribute ID Range. Each attribute ID Range is encoded as a 32-bit unsigned integer data element, where the high order 16 bits are interpreted as the beginning attribute ID of the range and the low order 16 bits are interpreted as the ending attribute ID of the range. The attribute IDs contained in the Attribute ID Ranges List must be listed in ascending order without duplication of any attribute ID values. Note that all attributes may be requested by specifying a range of 0x0000-0xFFFF. The *alen* parameter is the length of the Attribute ID Ranges List. The **SDP_ATTR_RANGE**(*lo*, *hi*) macro can be used to prepare Attribute ID Range. The *vp* parameter should be an array of *sdp_attr_t* structures. Each *sdp_attr_t* structure describes single SDP attribute and defined as follows:

```
struct sdp_attr {
        uint16_t        flags;
#define SDP_ATTR_OK             (0 << 0)
#define SDP_ATTR_INVALID        (1 << 0)
#define SDP_ATTR_TRUNCATED      (1 << 1)
        uint16_t        attr;  /* SDP_ATTR_xxx */
        uint32_t        vlen;  /* length of the value[] in bytes */
        uint8_t        *value; /* base pointer */
};
```

```
typedef struct sdp_attr         sdp_attr_t;
typedef struct sdp_attr *       sdp_attr_p;
```

The caller of the **sdp_search**() function is expected to prepare the array of *sdp_attr* structures and for each element of the array both *vlen* and *value* must be set. The **sdp_search**() function will fill each *sdp_attr_t* structure with attribute and value, i.e., it will set *flags*, *attr* and *vlen* fields. The actual value of the attribute will be copied into *value* buffer. Note: attributes are returned in the order they appear in the Service Search Attribute Response. SDP server could return several attributes for the same record. In this case the order of the attributes will be: all attributes for the first records, then all attributes for the secord record etc.

The **sdp_attr2desc**() and **sdp_uuid2desc**() functions each take a numeric attribute ID/UUID value and convert it to a human readable description.

The **sdp_register_service**() function is used to register service with the local SDP server. The *xss* parameter should point to a valid SDP session object obtained from **sdp_open_local**(). The *uuid* parameter is a SDP Service Class ID for the service to be registered. The *bdaddr* parameter should point to a valid BD_ADDR. The service will be only advertised if request was received by the local device with *bdaddr*. If *bdaddr* is set to BDADDR_ANY then the service will be advertised to any remote devices that queries for it. The *data* and *datalen* parameters specify data and size of the data for the service. Upon successful return **sdp_register_service**() will populate *handle* with the SDP record handle. This parameter is optional and can be set to NULL.

The **sdp_unregister_service**() function is used to unregister service with the local SDP server. The *xss* parameter should point to a valid SDP session object obtained from **sdp_open_local**(). The *handle* parameter should contain a valid SDP record handle of the service to be unregistered.

The **sdp_change_service**() function is used to change data associated with the existing service on the local SDP server. The *xss* parameter should point to a valid SDP session object obtained from **sdp_open_local**(). The *handle* parameter should contain a valid SDP record handle of the service to be changed. The *data* and *datalen* parameters specify data and size of the data for the service.

CAVEAT

When registering services with the local SDP server the application must keep the SDP session open. If SDP session is closed then the local SDP server will remove all services that were registered over the session. The application is allowed to change or unregister service if it was registered over the same session.

EXAMPLES

The following example shows how to get SDP_ATTR_PROTOCOL_DESCRIPTOR_LIST attribute for the SDP_SERVICE_CLASS_SERIAL_PORT service from the remote device.

```
bdaddr_t        remote;
uint8_t         buffer[1024];
void           *ss    = NULL;
uint16_t        serv  = SDP_SERVICE_CLASS_SERIAL_PORT;
uint32_t        attr  = SDP_ATTR_RANGE(
                              SDP_ATTR_PROTOCOL_DESCRIPTOR_LIST,
                              SDP_ATTR_PROTOCOL_DESCRIPTOR_LIST);
sdp_attr_t      proto = { SDP_ATTR_INVALID,0,sizeof(buffer),buffer };

/* Obtain/set remote BDADDR here */

if ((ss = sdp_open(BDADDR_ANY, remote)) == NULL)
        /* exit ENOMEM */
if (sdp_error(ss) != 0)
```

```
        /* exit sdp_error(ss) */

    if (sdp_search(ss, 1, &serv, 1, &attr, 1, &proto) != 0)
        /* exit sdp_error(ss) */

    if (proto.flags != SDP_ATTR_OK)
        /* exit see proto.flags for details */

    /* If we got here then we have attribute value in proto.value */
```

DIAGNOSTICS

Both **sdp_open**() and **sdp_open_local**() will return NULL if memory allocation for the new SDP session object fails. If the new SDP object was created then caller is still expected to call **sdp_error**() to check if there was connection error.

The **sdp_search**(), **sdp_register_service**(), **sdp_unregister_service**() and **sdp_change_service**() functions return non-zero value on error. The caller is expected to call **sdp_error**() to find out more about error.

SEE ALSO

bluetooth(3), sdpquery(1), sdpd(8), strerror(3)

AUTHORS

Maksim Yevmenkin ⟨m_evmenkin@yahoo.com⟩

BUGS

Most likely. Please report bugs if found.

HISTORY

libsdp first appeared in FreeBSD and was ported to NetBSD 4.0 by Iain Hibbert.

NAME

secure_path — determine if a file appears to be ‘‘secure’’

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
secure_path(const char *path);
```

DESCRIPTION

The **secure_path**() function takes a path name and returns zero if the referenced file is “secure”, non-zero if not. Any “insecurity”, other than failure to access the referenced file, will be logged to the system log.

To be “secure”, the referenced file must exist, be a regular file (and not a directory), owned by the super-user, and writable only by the super-user.

SEE ALSO

openlog(3)

HISTORY

The **secure_path** function is based on the BSD/OS implementation of same, and appeared in NetBSD 1.5 by kind permission.

NAME

sem_destroy — destroy an unnamed semaphore

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <semaphore.h>

int
sem_destroy(sem_t *sem);
```

DESCRIPTION

The **sem_destroy**() function destroys the unnamed semaphore pointed to by *sem*. After a successful call to **sem_destroy**(), *sem* is unusable until re-initialized by another call to **sem_init**().

RETURN VALUES

The **sem_destroy**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

sem_destroy() will fail if:

[EBUSY] There are currently threads blocked on the semaphore that *sem* points to.

[EINVAL] *sem* points to an invalid semaphore.

SEE ALSO

sem_init(3)

STANDARDS

sem_destroy() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

POSIX does not define the behavior of **sem_destroy**() if called while there are threads blocked on *sem*, but this implementation is guaranteed to return −1 and set *errno* to EBUSY if there are threads blocked on *sem*.

NAME

sem_getvalue — get the value of a semaphore

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <semaphore.h>

int
sem_getvalue(sem_t * restrict sem, int * restrict sval);
```

DESCRIPTION

The **sem_getvalue**() function sets the variable pointed to by *sval* to the current value of the semaphore pointed to by *sem*, as of the time that the call to **sem_getvalue**() is actually run.

RETURN VALUES

The **sem_getvalue**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

sem_getvalue() will fail if:

[EINVAL] *sem* points to an invalid semaphore.

SEE ALSO

sem_post(3), sem_trywait(3), sem_wait(3)

STANDARDS

sem_getvalue() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

The value of the semaphore is never negative, even if there are threads blocked on the semaphore. POSIX is somewhat ambiguous in its wording with regard to what the value of the semaphore should be if there are blocked waiting threads, but this behavior is conformant, given the wording of the specification.

NAME

sem_init — initialize an unnamed semaphore

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <semaphore.h>

int
sem_init(sem_t *sem, int pshared, unsigned int value);
```

DESCRIPTION

The **sem_init**() function initializes the unnamed semaphore pointed to by *sem* to have the value *value*. A non-zero value for *pshared* specifies a shared semaphore that can be used by multiple processes, which this implementation is not capable of.

Following a successful call to **sem_init**(), *sem* can be used as an argument in subsequent calls to *sem_wait*, *sem_trywait*, *sem_post*, and *sem_destroy*. *sem* is no longer valid after a successful call to *sem_destroy*.

RETURN VALUES

The **sem_init**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

sem_init() will fail if:

[EINVAL]	*value* exceeds SEM_VALUE_MAX.
[ENOSPC]	Memory allocation error.
[EPERM]	Unable to initialize a shared semaphore.

SEE ALSO

sem_destroy(3), sem_post(3), sem_trywait(3), sem_wait(3)

STANDARDS

sem_init() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

This implementation does not support shared semaphores, and reports this fact by setting *errno* to EPERM. This is perhaps a stretch of the intention of POSIX, but is compliant, with the caveat that **sem_init**() always reports a permissions error when an attempt to create a shared semaphore is made.

NAME

sem_open, **sem_close**, **sem_unlink** — named semaphore operations

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <semaphore.h>

sem_t *
sem_open(const char *name, int oflag, ...);

int
sem_close(sem_t *sem);

int
sem_unlink(const char *name);
```

DESCRIPTION

The **sem_open**() function creates or opens the named semaphore specified by *name*. The returned semaphore may be used in subsequent calls to sem_getvalue(3), sem_wait(3), sem_trywait(3), sem_post(3), and **sem_close**().

The following bits may be set in the *oflag* argument:

O_CREAT Create the semaphore if it does not already exist.

The third argument to the call to **sem_open**() must be of type *mode_t* and specifies the mode for the semaphore. Only the S_IWUSR, S_IWGRP, and S_IWOTH bits are examined; it is not possible to grant only "read" permission on a semaphore. The mode is modified according to the process's file creation mask; see umask(2).

The fourth argument must be an *unsigned int* and specifies the initial value for the semaphore, and must be no greater than SEM_VALUE_MAX.

O_EXCL Create the semaphore if it does not exist. If the semaphore already exists, **sem_open**() will fail. This flag is ignored unless O_CREAT is also specified.

The **sem_close**() function closes a named semaphore that was opened by a call to **sem_open**().

The **sem_unlink**() function removes the semaphore named *name*. Resources allocated to the semaphore are only deallocated when all processes that have the semaphore open close it.

RETURN VALUES

If successful, the **sem_open**() function returns the address of the opened semaphore. If the same *name* argument is given to multiple calls to **sem_open**() by the same process without an intervening call to **sem_close**(), the same address is returned each time. If the semaphore cannot be opened, **sem_open**() returns SEM_FAILED and the global variable *errno* is set to indicate the error.

The **sem_close**() and **sem_unlink**() functions return the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sem_open**() function will fail if:

[EACCES] The semaphore exists and the permissions specified by *oflag* at the time it was created deny access to this process.

[EACCES] The semaphore does not exist, but permission to create it is denied.

[EEXIST] O_CREAT and O_EXCL are set but the semaphore already exists.

[EINTR] The call was interrupted by a signal.

[EINVAL] The **sem_open**() operation is not supported for the given *name*.

[EINVAL] The *value* argument is greater than SEM_VALUE_MAX.

[ENAMETOOLONG] The *name* argument is too long.

[ENFILE] The system limit on semaphores has been reached.

[ENOENT] O_CREAT is not set and the named semaphore does not exist.

[ENOSPC] There is not enough space to create the semaphore.

The **sem_close**() function will fail if:

[EINVAL] The *sem* argument is not a valid semaphore.

The **sem_unlink**() function will fail if:

[EACCES] Permission is denied to unlink the semaphore.

[ENAMETOOLONG] The specified *name* is too long.

[ENOENT] The named semaphore does not exist.

SEE ALSO

close(2), open(2), umask(2), unlink(2), sem_getvalue(3), sem_post(3), sem_trywait(3), sem_wait(3), sem(4)

STANDARDS

The **sem_open**(), **sem_close**(), and **sem_unlink**() functions conform to ISO/IEC 9945-1:1996 ("POSIX.1").

HISTORY

Support for named semaphores first appeared in NetBSD 2.0.

BUGS

This implementation places strict requirements on the value of *name*: it must begin with a slash ('/'), contain no other slash characters, and be less than 14 characters in length not including the terminating null character.

NAME

sem_post — increment (unlock) a semaphore

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <semaphore.h>

int
sem_post(sem_t *sem);
```

DESCRIPTION

The **sem_post**() function increments (unlocks) the semaphore pointed to by *sem*. If there are threads blocked on the semaphore when **sem_post**() is called, then the highest priority thread that has been blocked the longest on the semaphore will be allowed to return from **sem_wait**().

sem_post() is signal-reentrant and may be called within signal handlers.

RETURN VALUES

The **sem_post**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

sem_post() will fail if:

[EINVAL] *sem* points to an invalid semaphore.

SEE ALSO

sem_trywait(3), sem_wait(3)

STANDARDS

sem_post() conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

sem_wait, **sem_trywait** — decrement (lock) a semaphore

LIBRARY

POSIX Real-time Library (librt, −lrt)

SYNOPSIS

```
#include <semaphore.h>

int
sem_wait(sem_t *sem);

int
sem_trywait(sem_t *sem);
```

DESCRIPTION

The **sem_wait**() function decrements (locks) the semaphore pointed to by *sem*, but blocks if the value of *sem* is zero, until the value is non-zero and the value can be decremented.

The **sem_trywait**() function decrements (locks) the semaphore pointed to by *sem* only if the value is non-zero. Otherwise, the semaphore is not decremented and an error is returned.

RETURN VALUES

The **sem_wait**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

sem_wait() and **sem_trywait**() will fail if:

[EINVAL] *sem* points to an invalid semaphore.

Additionally, **sem_trywait**() will fail if:

[EAGAIN] The semaphore value was zero, and thus could not be decremented.

SEE ALSO

sem_post(3)

STANDARDS

sem_wait() and **sem_trywait**() conform to ISO/IEC 9945-1:1996 (“POSIX.1”).

NAME

setbuf, **setbuffer**, **setlinebuf**, **setvbuf** — stream buffering operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

void
setbuf(FILE * restrict stream, char * restrict buf);

void
setbuffer(FILE *stream, char *buf, size_t size);

int
setlinebuf(FILE *stream);

int
setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);
```

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a newline is output or input is read from any stream attached to a terminal device (typically stdin). The function `fflush`(3) may be used to force the block out early. (See `fclose`(3).)

Normally all files are block buffered. When the first I/O operation occurs on a file, `malloc`(3) is called, and an optimally-sized buffer is obtained. If a stream refers to a terminal (as *stdout* normally does) it is line buffered. The standard error stream *stderr* is initially unbuffered.

The **setvbuf**() function may be used to alter the buffering behavior of a stream. The *mode* parameter must be one of the following three macros:

`_IONBF` unbuffered

`_IOLBF` line buffered

`_IOFBF` fully buffered

The *size* parameter may be given as zero to obtain deferred optimal-size buffer allocation as usual. If it is not zero, then except for unbuffered files, the *buf* argument should point to a buffer at least *size* bytes long; this buffer will be used instead of the current buffer. (If the *size* argument is not zero but *buf* is `NULL`, a buffer of the given size will be allocated immediately, and released on close. This is an extension to ANSI C; portable code should use a size of 0 with any `NULL` buffer.)

The **setvbuf**() function may be used at any time, but may have peculiar side effects (such as discarding input or flushing output) if the stream is ‘‘active’’. Portable applications should call it only once on any given stream, and before any I/O is performed.

The other three calls are, in effect, simply aliases for calls to **setvbuf**(). Except for the lack of a return value, the **setbuf**() function is exactly equivalent to the call

```
setvbuf(stream, buf, buf ? _IOFBF : _IONBF, BUFSIZ);
```

The **setbuffer**() function is the same, except that the size of the buffer is up to the caller, rather than being determined by the default `BUFSIZ`. The **setlinebuf**() function is exactly equivalent to the call:

```
setvbuf(stream, (char *)NULL, _IOLBF, 0);
```

RETURN VALUES

The **setvbuf**() function returns 0 on success, or EOF if the request cannot be honored (note that the stream is still functional in this case).

The **setlinebuf**() function returns what the equivalent **setvbuf**() would have returned.

SEE ALSO

fclose(3), fopen(3), fread(3), malloc(3), printf(3), puts(3)

STANDARDS

The **setbuf**() and **setvbuf**() functions conform to ANSI X3.159-1989 ("ANSI C89").

BUGS

The **setbuffer**() and **setlinebuf**() functions are not portable to versions of BSD before 4.2BSD. On 4.2BSD and 4.3BSD systems, **setbuf**() always uses a suboptimal buffer size and should be avoided.

NAME

sigsetjmp, **siglongjmp**, **setjmp**, **longjmp**, **_setjmp**, **_longjmp**, **longjmperror** — non-local jumps

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <setjmp.h>

int
sigsetjmp(sigjmp_buf env, int savemask);

void
siglongjmp(sigjmp_buf env, int val);

int
setjmp(jmp_buf env);

void
longjmp(jmp_buf env, int val);

int
_setjmp(jmp_buf env);

void
_longjmp(jmp_buf env, int val);

void
longjmperror(void);
```

DESCRIPTION

The **sigsetjmp**(), **setjmp**(), and **_setjmp**() functions save their calling environment in *env*. Each of these functions returns 0.

The corresponding **longjmp**() functions restore the environment saved by the most recent invocation of the respective **setjmp**() function. They then return so that program execution continues as if the corresponding invocation of the **setjmp**() call had just returned the value specified by *val*, instead of 0.

Pairs of calls may be intermixed, i.e., both **sigsetjmp**() and **siglongjmp**() as well as **setjmp**() and **longjmp**() combinations may be used in the same program. However, individual calls may not, e.g., the *env* argument to **setjmp**() may not be passed to **siglongjmp**().

The **longjmp**() routines may not be called after the routine which called the **setjmp**() routines returns.

All accessible objects have values as of the time **longjmp**() routine was called, except that the values of objects of automatic storage invocation duration that do not have the `volatile` type and have been changed between the **setjmp**() invocation and **longjmp**() call are indeterminate.

The **setjmp**()/**longjmp**() function pairs save and restore the signal mask while **_setjmp**()/**_longjmp**() function pairs save and restore only the register set and the stack. (See **sigprocmask**(*2*).)

The **sigsetjmp**()/**siglongjmp**() function pairs save and restore the signal mask if the argument *savemask* is non-zero. Otherwise, only the register set and the stack are saved.

In other words, **setjmp**()/**longjmp**() are functionally equivalent to **sigsetjmp**()/**siglongjmp**() when **sigsetjmp**() is called with a non-zero *savemask* argument. Conversely, **_setjmp**()/**_longjmp**() are functionally equivalent to **sigsetjmp**()/**siglongjmp**() when **sigsetjmp**() is called with a zero-value *savemask*.

The **sigsetjmp**()/**siglongjmp**() interfaces are preferred for maximum portability.

ERRORS

If the contents of the *env* are corrupted or correspond to an environment that has already returned, the **longjmp**() routine calls the routine longjmperror(3). If **longjmperror**() returns, the program is aborted (see abort(3)). The default version of **longjmperror**() prints the message "longjmp botch" to standard error and returns. User programs wishing to exit more gracefully should write their own versions of **longjmperror**().

SEE ALSO

sigaction(2), sigaltstack(2), sigprocmask(2), pthread_sigmask(3), signal(3)

STANDARDS

The **setjmp**() and **longjmp**() functions conform to ANSI X3.159-1989 ("ANSI C89"). The **sigsetjmp**() and **siglongjmp**() functions conform to ISO/IEC 9945-1:1990 ("POSIX.1").

CAVEATS

Historically, on AT&T System V UNIX, the **setjmp**()/**longjmp**() functions have been equivalent to the BSD **_setjmp**()/**_longjmp**() functions and do not restore the signal mask. Because of this discrepancy, the **sigsetjmp**()/**siglongjmp**() interfaces should be used if portability is desired.

Use of **longjmp**() or **siglongjmp**() from inside a signal handler is not as easy as it might seem. Generally speaking, all possible code paths between the **setjmp**() and **longjmp**() must be signal race safe. Furthermore, the code paths must not do resource management (such as open(2) or close(2)) without blocking the signal in question, or resources might be mismanaged. Obviously this makes **longjmp**() much less useful than previously thought.

NAME

setlocale, **localeconv** — natural language formatting for C

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <locale.h>

char *
setlocale(int category, const char *locale);

struct lconv *
localeconv(void);
```

DESCRIPTION

The **setlocale**() function sets the C library's notion of natural language formatting style for particular sets of routines. Each such style is called a 'locale' and is invoked using an appropriate name passed as a C string. The **localeconv**() routine returns the current locale's parameters for formatting numbers.

The **setlocale**() function recognizes several categories of routines. These are the categories and the sets of routines they select:

LC_ALL — Set the entire locale generically.

LC_COLLATE — Set a locale for string collation routines. This controls alphabetic ordering in **strcoll**() and **strxfrm**().

LC_CTYPE — Set a locale for the ctype(3) functions. This controls recognition of upper and lower case, alphabetic or non-alphabetic characters, and so on. The real work is done by the **setrunelocale**() function.

LC_MESSAGES — Set a locale for message catalogs. This controls the selection of message catalogs by the catgets(3) and gettext(3) families of functions.

LC_MONETARY — Set a locale for formatting monetary values; this affects the **localeconv**() function.

LC_NUMERIC — Set a locale for formatting numbers. This controls the formatting of decimal points in input and output of floating point numbers in functions such as **printf**() and **scanf**(), as well as values returned by **localeconv**().

LC_TIME — Set a locale for formatting dates and times using the **strftime**() function.

Only three locales are defined by default, the empty string "" which denotes the native environment, and the "C" and "POSIX" locales, which denote the C language environment. A *locale* argument of NULL causes **setlocale**() to return the current locale. By default, C programs start in the "C" locale. The format of the locale string is described in nls(7).

The only function in the library that sets the locale is **setlocale**(); the locale is never changed as a side effect of some other routine.

Changing the setting of LC_MESSAGES has no effect on catalogs that have already been opened by catopen(3).

The **localeconv**() function returns a pointer to a structure which provides parameters for formatting numbers, especially currency values:

```
struct lconv {
        char   *decimal_point;
```

```
        char    *thousands_sep;
        char    *grouping;
        char    *int_curr_symbol;
        char    *currency_symbol;
        char    *mon_decimal_point;
        char    *mon_thousands_sep;
        char    *mon_grouping;
        char    *positive_sign;
        char    *negative_sign;
        char    int_frac_digits;
        char    frac_digits;
        char    p_cs_precedes;
        char    p_sep_by_space;
        char    n_cs_precedes;
        char    n_sep_by_space;
        char    p_sign_posn;
        char    n_sign_posn;
        char    int_p_cs_precedes;
        char    int_n_cs_precedes;
        char    int_p_sep_by_space;
        char    int_n_sep_by_space;
        char    int_p_sign_posn;
        char    int_n_sign_posn;
};
```

The individual fields have the following meanings:

decimal_point The decimal point character, except for monetary values.

thousands_sep The separator between groups of digits before the decimal point, except for monetary values.

grouping The sizes of the groups of digits, except for monetary values. This is a pointer to a vector of integers, each of size *char*, representing group size from low order digit groups to high order (right to left). The list may be terminated with 0 or CHAR_MAX. If the list is terminated with 0, the last group size before the 0 is repeated to account for all the digits. If the list is terminated with CHAR_MAX, no more grouping is performed.

int_curr_symbol The standardized (ISO 4217:1995) international currency symbol.

currency_symbol The local currency symbol.

mon_decimal_point The decimal point character for monetary values.

mon_thousands_sep The separator for digit groups in monetary values.

mon_grouping Like *grouping* but for monetary values.

positive_sign The character used to denote nonnegative monetary values, usually the empty string.

negative_sign The character used to denote negative monetary values, usually a minus sign.

int_frac_digits The number of digits after the decimal point in an internationally formatted monetary value.

frac_digits The number of digits after the decimal point in an locally formatted monetary value.

p_cs_precedes 1 if the currency symbol precedes the monetary value for nonnegative values, 0 if it follows.

p_sep_by_space 1 if a space is inserted between the currency symbol and the monetary value for nonnegative values, 0 otherwise.

n_cs_precedes Like *p_cs_precedes* but for negative values.

n_sep_by_space Like *p_sep_by_space* but for negative values.

p_sign_posn The location of the *positive_sign* with respect to a nonnegative quantity and the *currency_symbol*.

n_sign_posn Like *p_sign_posn* but for negative currency values.

int_p_cs_precedes 1 if the currency symbol precedes the internationally formatted monetary value for nonnegative values, 0 if it follows.

int_n_cs_precedes Like *int_p_cs_precedes* but for negative values.

int_p_sep_by_space 1 if a space is inserted between the currency symbol and the internationally formatted monetary value for nonnegative values, 0 otherwise.

int_n_sep_by_space Like *int_p_sep_by_space* but for negative values.

int_p_sign_posn The location of the *positive_sign* with respect to a nonnegative quantity and the *currency_symbol*, for internationally formatted nonnegative monetary values.

int_n_sign_posn Like *int_p_sign_posn* but for negative values.

The positional parameters in *p_sign_posn*, *n_sign_posn*, *int_p_sign_posn* and *int_n_sign_posn* are encoded as follows:

0 Parentheses around the entire string.
1 Before the string.
2 After the string.
3 Just before *currency_symbol*.
4 Just after *currency_symbol*.

Unless mentioned above, an empty string as a value for a field indicates a zero length result or a value that is not in the current locale. A CHAR_MAX result similarly denotes an unavailable value.

RETURN VALUES

The **setlocale**() function returns NULL and fails to change the locale if the given combination of *category* and *locale* makes no sense. The **localeconv**() function returns a pointer to a static object which may be altered by later calls to **setlocale**() or **localeconv**().

EXAMPLES

The following code illustrates how a program can initialize the international environment for one language, while selectively modifying the program's locale such that regular expressions and string operations can be applied to text recorded in a different language:

```
setlocale(LC_ALL, "de");
setlocale(LC_COLLATE, "fr");
```

When a process is started, its current locale is set to the C or POSIX locale. An internationalized program that depends on locale data not defined in the C or POSIX locale must invoke the setlocale subroutine in the following manner before using any of the locale-specific information:

```
setlocale(LC_ALL, "");
```

SEE ALSO

`catopen`(3), `gettext`(3), `nl_langinfo`(3), `nls`(7)

STANDARDS

The **`setlocale`**() and **`localeconv`**() functions conform to ANSI X3.159-1989 ("ANSI C89") and ISO/IEC 9899:1990 ("ISO C90").

The *`int_p_cs_precedes`*, *`int_n_cs_precedes`*, *`int_p_sep_by_space`*, *`int_n_sep_by_space`*, *`int_p_sign_posn`* and *`int_n_sign_posn`* members of *`struct lconv`* were introduced in ISO/IEC 9899:1999 ("ISO C99").

HISTORY

The **`setlocale`**() and **`localeconv`**() functions first appeared in 4.4BSD.

BUGS

The current implementation supports only the `"C"` and `"POSIX"` locales for all but the `LC_CTYPE` locale.

In spite of the gnarly currency support in **`localeconv`**(), the standards don't include any functions for generalized currency formatting.

`LC_COLLATE` does not make sense for many languages. Use of `LC_MONETARY` could lead to misleading results until we have a real time currency conversion function. `LC_NUMERIC` and `LC_TIME` are personal choices and should not be wrapped up with the other categories.

Multibyte locales aren't supported for static binaries.

NAME

getmode, **setmode** — modify mode bits

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

mode_t
getmode(const void *set, mode_t mode);

void *
setmode(const char *mode_str);
```

DESCRIPTION

The **getmode**() function returns a copy of the file permission bits *mode* as altered by the values pointed to by *set*. While only the mode bits are altered, other parts of the file mode may be examined.

The **setmode**() function takes an absolute (octal) or symbolic value, as described in chmod(1), as an argument and returns a pointer to mode values to be supplied to **getmode**(). Because some of the symbolic values are relative to the file creation mask, **setmode**() may call umask(2). If this occurs, the file creation mask will be restored before **setmode**() returns. If the calling program changes the value of its file creation mask after calling **setmode**(), **setmode**() must be called again if **getmode**() is to modify future file modes correctly.

If the mode passed to **setmode**() is invalid, **setmode**() returns NULL.

ERRORS

The **setmode**() function may fail and set *errno* for any of the errors specified for the library routines malloc(3) or strtol(3). In addition, **setmode**() will fail and set *errno* to:

[EINVAL] The *mode* argument does not represent a valid mode.

SEE ALSO

chmod(1), stat(2), umask(2), malloc(3)

HISTORY

The **getmode**() and **setmode**() functions first appeared in 4.4BSD.

NAME

setproctitle — set process title

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

void
setproctitle(const char *fmt, ...);
```

DESCRIPTION

The **setproctitle**() function sets the invoking process's title. The process title is set to the last component of the program name, followed by a colon and the formatted string specified by *fmt*. If *fmt* is NULL, the colon and formatted string are omitted. The length of a process title is limited to 2048 bytes.

EXAMPLES

Set the process title to the program name, with no further information:

```
setproctitle(NULL);
```

Set the process title to the program name, an informational string, and the process id:

```
setproctitle("foo! (%d)", getpid());
```

SEE ALSO

ps(1), w(1), getprogname(3), printf(3)

HISTORY

The **setproctitle**() function first appeared in NetBSD 1.0.

CAVEATS

It is important never to pass a string with user-supplied data as a format without using '%s'. An attacker can put format specifiers in the string to mangle your stack, leading to a possible security hole. This holds true even if you have built the string "by hand" using a function like **snprintf**(), as the resulting string may still contain user-supplied conversion specifiers for later interpolation by **setproctitle**().

Always be sure to use the proper secure idiom:

```
setproctitle("%s", string);
```

NAME

setruid, **setrgid** — set user and group ID

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>

int
setruid(uid_t ruid);

int
setrgid(gid_t rgid);
```

DESCRIPTION

The **setruid**() function (**setrgid**()) sets the real user ID (group ID) of the current process.

RETURN VALUES

Upon success, these functions return 0; otherwise −1 is returned.

If the user is not the super user, or the uid specified is not the real or effective ID, these functions return −1.

The use of these calls is not portable. Their use is discouraged; they will be removed in the future.

SEE ALSO

getgid(2), getuid(2), setegid(2), seteuid(2), setgid(2), setuid(2)

HISTORY

The **setruid**() and **setrgid**() syscalls appeared in 4.2BSD and were dropped in 4.4BSD.

NAME

SHA1Init, **SHA1Update**, **SHA1Final**, **SHA1Transform**, **SHA1End**, **SHA1File**, **SHA1Data** — calculate the NIST Secure Hash Algorithm

SYNOPSIS

```
#include <sys/types.h>
#include <sha1.h>

void
SHA1Init(SHA1_CTX *context);

void
SHA1Update(SHA1_CTX *context, const u_char *data, u_int len);

void
SHA1Final(u_char digest[20], SHA1_CTX *context);

void
SHA1Transform(uint32_t state[5], u_char buffer[64]);

char *
SHA1End(SHA1_CTX *context, char *buf);

char *
SHA1File(char *filename, char *buf);

char *
SHA1Data(u_char *data, size_t len, char *buf);
```

DESCRIPTION

The SHA1 functions implement the NIST Secure Hash Algorithm (SHA-1), FIPS PUB 180-1. SHA-1 is used to generate a condensed representation of a message called a message digest. The algorithm takes a message less than 2^64 bits as input and produces a 160-bit digest suitable for use as a digital signature.

The SHA1 functions are considered to be more secure than the md4(3) and md5(3) functions with which they share a similar interface.

The **SHA1Init**() function initializes a SHA1_CTX *context* for use with **SHA1Update**(), and **SHA1Final**(). The **SHA1Update**() function adds *data* of length *len* to the SHA1_CTX specified by *context*. **SHA1Final**() is called when all data has been added via **SHA1Update**() and stores a message digest in the *digest* parameter. When a null pointer is passed to **SHA1Final**() as first argument only the final padding will be applied and the current context can still be used with **SHA1Update**().

The **SHA1Transform**() function is used by **SHA1Update**() to hash 512-bit blocks and forms the core of the algorithm. Most programs should use the interface provided by **SHA1Init**(), **SHA1Update**() and **SHA1Final**() instead of calling **SHA1Transform**() directly.

The **SHA1End**() function is a front end for **SHA1Final**() which converts the digest into an ASCII representation of the 160 bit digest in hexadecimal.

The **SHA1File**() function calculates the digest for a file and returns the result via **SHA1End**(). If **SHA1File**() is unable to open the file a NULL pointer is returned.

The **SHA1Data**() function calculates the digest of an arbitrary string and returns the result via **SHA1End**().

For each of the **SHA1End**(), **SHA1File**(), and **SHA1Data**() functions the *buf* parameter should either be a string of at least 41 characters in size or a NULL pointer. In the latter case, space will be dynamically allocated via malloc(3) and should be freed using free(3) when it is no longer needed.

EXAMPLES

The follow code fragment will calculate the digest for the string "abc" which is ‘‘0xa9993e36476816aba3e25717850c26c9cd0d89d’’.

```
SHA1_CTX sha;
u_char results[20];
char *buf;
int n;

buf = "abc";
n = strlen(buf);
SHA1Init(&sha);
SHA1Update(&sha, (u_char *)buf, n);
SHA1Final(results, &sha);

/* Print the digest as one long hex value */
printf("0x");
for (n = 0; n < 20; n++)
        printf("%02x", results[n]);
putchar('\n');
```

Alternately, the helper functions could be used in the following way:

```
SHA1_CTX sha;
u_char output[41];
char *buf = "abc";

printf("0x%s", SHA1Data(buf, strlen(buf), output));
```

SEE ALSO

md5(1), md4(3), md5(3)

J. Burrows, *The Secure Hash Standard*, FIPS PUB 180-1.

HISTORY

The SHA-1 functions appeared in NetBSD 1.4.

AUTHORS

This implementation of SHA-1 was written by Steve Reid.

The **SHA1End**(), **SHA1File**(), and **SHA1Data**() helper functions are derived from code written by Poul-Henning Kamp.

BUGS

This implementation of SHA-1 has not been validated by NIST and as such is not in official compliance with the standard.

If a message digest is to be copied to a multi-byte type (ie: an array of five 32-bit integers) it will be necessary to perform byte swapping on little endian machines such as the i386, alpha, and VAX.

NAME

SHA256_Init, **SHA256_Update**, **SHA256_Pad**, **SHA256_Final**, **SHA256_Transform**, **SHA256_End**, **SHA256_File**, **SHA256_FileChunk**, **SHA256_Data** — calculate the NIST Secure Hash Standard (version 2)

SYNOPSIS

```
#include <sys/types.h>
#include <sha2.h>

void
SHA256_Init(SHA256_CTX *context);

void
SHA256_Update(SHA256_CTX *context, const uint8_t *data, size_t len);

void
SHA256_Pad(SHA256_CTX *context);

void
SHA256_Final(uint8_t digest[SHA256_DIGEST_LENGTH], SHA256_CTX *context);

void
SHA256_Transform(uint32_t state[8],
    const uint8_t buffer[SHA256_BLOCK_LENGTH]);

char *
SHA256_End(SHA256_CTX *context, char *buf);

char *
SHA256_File(const char *filename, char *buf);

char *
SHA256_FileChunk(const char *filename, char *buf, off_t offset,
    off_t length);

char *
SHA256_Data(uint8_t *data, size_t len, char *buf);

void
SHA384_Init(SHA384_CTX *context);

void
SHA384_Update(SHA384_CTX *context, const uint8_t *data, size_t len);

void
SHA384_Pad(SHA384_CTX *context);

void
SHA384_Final(uint8_t digest[SHA384_DIGEST_LENGTH], SHA384_CTX *context);

void
SHA384_Transform(uint64_t state[8],
    const uint8_t buffer[SHA384_BLOCK_LENGTH]);

char *
SHA384_End(SHA384_CTX *context, char *buf);

char *
SHA384_File(char *filename, char *buf);
```

char ∗
SHA384_FileChunk(*char ∗filename*, *char ∗buf*, *off_t offset*, *off_t length*);

char ∗
SHA384_Data(*uint8_t ∗data*, *size_t len*, *char ∗buf*);

void
SHA512_Init(*SHA512_CTX ∗context*);

void
SHA512_Update(*SHA512_CTX ∗context*, *const uint8_t ∗data*, *size_t len*);

void
SHA512_Pad(*SHA512_CTX ∗context*);

void
SHA512_Final(*uint8_t digest[SHA512_DIGEST_LENGTH]*, *SHA512_CTX ∗context*);

void
SHA512_Transform(*uint64_t state[8]*,
const uint8_t buffer[SHA512_BLOCK_LENGTH]);

char ∗
SHA512_End(*SHA512_CTX ∗context*, *char ∗buf*);

char ∗
SHA512_File(*char ∗filename*, *char ∗buf*);

char ∗
SHA512_FileChunk(*char ∗filename*, *char ∗buf*, *off_t offset*, *off_t length*);

char ∗
SHA512_Data(*uint8_t ∗data*, *size_t len*, *char ∗buf*);

DESCRIPTION

The SHA2 functions implement the NIST Secure Hash Standard, FIPS PUB 180-2. The SHA2 functions are used to generate a condensed representation of a message called a message digest, suitable for use as a digital signature. There are three families of functions, with names corresponding to the number of bits in the resulting message digest. The SHA-256 functions are limited to processing a message of less than 2^64 bits as input. The SHA-384 and SHA-512 functions can process a message of at most 2^128 - 1 bits as input.

The SHA2 functions are considered to be more secure than the sha1(3) functions with which they share a similar interface. The 256, 384, and 512-bit versions of SHA2 share the same interface. For brevity, only the 256-bit variants are described below.

The **SHA256_Init**() function initializes a SHA256_CTX *context* for use with **SHA256_Update**(), and **SHA256_Final**(). The **SHA256_Update**() function adds *data* of length *len* to the SHA256_CTX specified by *context*. **SHA256_Final**() is called when all data has been added via **SHA256_Update**() and stores a message digest in the *digest* parameter.

The **SHA256_Pad**() function can be used to apply padding to the message digest as in **SHA256_Final**(), but the current context can still be used with **SHA256_Update**().

The **SHA256_Transform**() function is used by **SHA256_Update**() to hash 512-bit blocks and forms the core of the algorithm. Most programs should use the interface provided by **SHA256_Init**(), **SHA256_Update**(), and **SHA256_Final**() instead of calling **SHA256_Transform**() directly.

The **SHA256_End**() function is a front end for **SHA256_Final**() which converts the digest into an ASCII representation of the digest in hexadecimal.

The **SHA256_File**() function calculates the digest for a file and returns the result via **SHA256_End**(). If **SHA256_File**() is unable to open the file, a NULL pointer is returned.

SHA256_FileChunk() behaves like **SHA256_File**() but calculates the digest only for that portion of the file starting at *offset* and continuing for *length* bytes or until end of file is reached, whichever comes first. A zero *length* can be specified to read until end of file. A negative *length* or *offset* will be ignored.

The **SHA256_Data**() function calculates the digest of an arbitrary string and returns the result via **SHA256_End**().

For each of the **SHA256_End**(), **SHA256_File**(), **SHA256_FileChunk**(), and **SHA256_Data**() functions the *buf* parameter should either be a string large enough to hold the resulting digest (e.g., SHA256_DIGEST_STRING_LENGTH, SHA384_DIGEST_STRING_LENGTH, or SHA512_DIGEST_STRING_LENGTH, depending on the function being used) or a NULL pointer. In the latter case, space will be dynamically allocated via malloc(3) and should be freed using free(3) when it is no longer needed.

EXAMPLES

The following code fragment will calculate the SHA-256 digest for the string "abc", which is “0xba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad”.

```
SHA256_CTX ctx;
uint8_t results[SHA256_DIGEST_LENGTH];
char *buf;
int n;

buf = "abc";
n = strlen(buf);
SHA256_Init(&ctx);
SHA256_Update(&ctx, (uint8_t *)buf, n);
SHA256_Final(results, &ctx);

/* Print the digest as one long hex value */
printf("0x");
for (n = 0; n < SHA256_DIGEST_LENGTH; n++)
        printf("%02x", results[n]);
putchar('\n');
```

Alternately, the helper functions could be used in the following way:

```
SHA256_CTX ctx;
uint8_t output[SHA256_DIGEST_STRING_LENGTH];
char *buf = "abc";

printf("0x%s\n", SHA256_Data(buf, strlen(buf), output));
```

SEE ALSO

cksum(1), md4(3), md5(3), rmd160(3), sha1(3)

Secure Hash Standard, FIPS PUB 180-2.

HISTORY

The SHA2 functions appeared in OpenBSD 3.4 and NetBSD 3.0.

AUTHORS

This implementation of the SHA functions was written by Aaron D. Gifford.

The **SHA256_End**(), **SHA256_File**(), **SHA256_FileChunk**(), and **SHA256_Data**() helper functions are derived from code written by Poul-Henning Kamp.

CAVEATS

This implementation of the Secure Hash Standard has not been validated by NIST and as such is not in official compliance with the standard.

If a message digest is to be copied to a multi-byte type (i.e.: an array of five 32-bit integers) it will be necessary to perform byte swapping on little endian machines such as the i386, alpha, and vax.

NAME

shquote, **shquotev** — quote argument strings for use with the shell

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

size_t
shquote(const char *arg, char *buf, size_t bufsize);

size_t
shquotev(int argc, char * const *argv, char *buf, size_t bufsize);
```

DESCRIPTION

The **shquote**() and **shquotev**() functions copy strings and transform the copies by adding shell escape and quoting characters. They are used to encapsulate arguments to be included in command strings passed to the **system**() and **popen**() functions, so that the arguments will have the correct values after being evaluated by the shell.

The exact method of quoting and escaping may vary, and is intended to match the conventions of the shell used by **system**() and **popen**(). It may not match the conventions used by other shells. In this implementation, the following transformation is applied to each input string:

- it is surrounded by single quotes (`'`),
- any single quotes in the input are escaped by replacing them with the four-character sequence: `'\''`, and
- extraneous pairs of single quotes (caused by multiple adjacent single quotes in the input string, or by single quotes at the beginning or end of the input string) are elided.

The **shquote**() function transforms the string specified by its *arg* argument, and places the result into the memory pointed to by *buf*.

The **shquotev**() function transforms each of the *argc* strings specified by the array *argv* independently. The transformed strings are placed in the memory pointed to by *buf*, separated by spaces. It does not modify the pointer array specified by *argv* or the strings pointed to by the pointers in the array.

Both functions write up to *bufsize* - 1 characters of output into the buffer pointed to by *buf*, then add a `NUL` character to terminate the output string. If *bufsize* is given as zero, the *buf* parameter is ignored and no output is written.

RETURN VALUES

The **shquote**() and **shquotev**() functions return the number of characters necessary to hold the result from operating on their input strings, not including the terminating `NUL`. That is, they return the length of the string that would have been written to the output buffer, if it were large enough. If an error occurs during processing, the value ((size_t)−1) is returned and *errno* is set appropriately.

EXAMPLES

The following code fragment demonstrates how you might use **shquotev**() to construct a command string to be used with **system**(). The command uses an environment variable (which will be expanded by the shell) to determine the actual program to run. Note that the environment variable may be expanded by the shell into multiple words. The first word of the expansion will be used by the shell as the name of the program to run, and the rest will be passed as arguments to the program.

```
char **argv, c, *cmd;
size_t cmdlen, len, qlen;
int argc;

...

/*
 * Size buffer to hold the command string, and allocate it.
 * Buffer of length one given to snprintf() for portability.
 */
cmdlen = snprintf(&c, 1, "${PROG-%s} ", PROG_DEFAULT);
qlen = shquotev(argc, argv, NULL, 0) + 1;
if (qlen == (size_t)-1) {
        ...
}
cmdlen += qlen;
cmd = malloc(cmdlen);
if (cmd == NULL) {
        ...
}

/* Create the command string. */
len = snprintf(cmd, cmdlen, "${PROG-%s} ", PROG_DEFAULT);
qlen = shquotev(argc, argv, cmd + len, cmdlen - len);
if (qlen == (size_t)-1) {
        /* Should not ever happen. */
        ...
}
len += qlen;

/* "cmd" can now be passed to system(). */
```

The following example shows how you would implement the same functionality using the **shquote**() function directly.

```
char **argv, c, *cmd;
size_t cmdlen, len, qlen;
int argc, i;

...

/*
 * Size buffer to hold the command string, and allocate it.
 * Buffer of length one given to snprintf() for portability.
 */
cmdlen = snprintf(&c, 1, "${PROG-%s} ", PROG_DEFAULT);
for (i = 0; i < argc; i++) {
        qlen = shquote(argv[i], NULL, 0) + 1;
        if (qlen == (size_t)-1) {
                ...
        }
        cmdlen += qlen;
}
```

```
cmd = malloc(cmdlen);
if (cmd == NULL) {
        ...
}

/* Start the command string with the env var reference. */
len = snprintf(cmd, cmdlen, "${PROG-%s} ", PROG_DEFAULT);

/* Quote all of the arguments when copying them. */
for (i = 0; i < argc; i++) {
        qlen = shquote(argv[i], cmd + len, cmdlen - len);
        if (qlen == (size_t)-1) {
                /* Should not ever happen. */
                ...
        }
        len += qlen;
        cmd[len++] = ' ';
}
cmd[--len] = ' ';

/* "cmd" can now be passed to system(). */
```

SEE ALSO

sh(1), popen(3), system(3)

BUGS

This implementation does not currently handle strings containing multibyte characters properly. To address this issue, /bin/sh (the shell used by **system**() and **popen**()) must first be fixed to handle multibyte characters. When that has been done, these functions can have multibyte character support enabled.

NAME

sigblock — block signals

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

int
sigblock(int mask);

int
sigmask(signum);
```

DESCRIPTION

This interface is made obsolete by: sigprocmask(**2**)**.**

sigblock() adds the signals specified in *mask* to the set of signals currently being blocked from delivery. Signals are blocked if the corresponding bit in *mask* is a 1; the macro **sigmask**() is provided to construct the mask for a given *signum*.

It is not possible to block SIGKILL or SIGSTOP; this restriction is silently imposed by the system.

RETURN VALUES

The previous set of masked signals is returned.

EXAMPLES

The following example using **sigblock**():

```
int omask;

omask = sigblock(sigmask(SIGINT) | sigmask(SIGHUP));
```

Becomes:

```
sigset_t set, oset;

sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGHUP);
sigprocmask(SIG_BLOCK, &set, &oset);
```

Another use of **sigblock**() is to get the current set of masked signals without changing what is actually blocked. Instead of:

```
int set;

set = sigblock(0);
```

Use the following:

```
sigset_t set;

sigprocmask(SIG_BLOCK, NULL, &set);
```

SEE ALSO

`kill`(2), `sigaction`(2), `sigprocmask`(2), `sigsetmask`(3), `sigsetops`(3)

HISTORY

The **`sigblock`**() function call appeared in 4.2BSD and has been deprecated.

NAME

sighold — manipulate current signal mask

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

int
sighold(int sig);
```

DESCRIPTION

This interface is made obsolete by `sigprocmask`**(2).**

The **sighold**() function adds the signal *sig* to the calling process' signal mask.

RETURN VALUES

If successful, the **sighold**() function returns 0. Otherwise −1 is returned and *errno* is set to indicate the error.

ERRORS

The **sighold**() function will fail if:

`[EINVAL]` The argument *sig* is not a valid signal number.

SEE ALSO

`sigprocmask`(2), `sigrelse`(3)

STANDARDS

The **sighold**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

sigignore — manipulate signal dispositions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

int
sigignore(int sig);
```

DESCRIPTION

This interface is made obsolete by sigaction(**2**)**.**

The **sigignore**() function sets the disposition of the signal *sig* to SIG_IGN.

RETURN VALUES

If successful, the **sigignore**() function returns 0. Otherwise −1 is returned and *errno* is set to indicate the error.

ERRORS

The **sigignore**() function will fail if:

[EINVAL] The argument *sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore a signal that cannot be ignored, such as SIGKILL or SIGSTOP.

SEE ALSO

sigaction(2)

STANDARDS

The **sigignore**() function conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

siginterrupt — allow signals to interrupt system calls

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

int
siginterrupt(int sig, int flag);
```

DESCRIPTION

The **siginterrupt**() function is used to change the system call restart behavior when a system call is interrupted by the specified signal. If the flag is false (0), then system calls will be restarted if they are interrupted by the specified signal and no data has been transferred yet. System call restart is the default behavior on 4.2BSD.

If the flag is true (1), then restarting of system calls is disabled. If a system call is interrupted by the specified signal and no data has been transferred, the system call will return −1 with the global variable *errno* set to EINTR. Interrupted system calls that have started transferring data will return the amount of data actually transferred. System call interrupt is the signal behavior found on 4.1BSD and AT&T System V UNIX systems.

Note that the new 4.2BSD signal handling semantics are not altered in any other way. Most notably, signal handlers always remain installed until explicitly changed by a subsequent sigaction(2) call, and the signal mask operates as documented in sigaction(2). Programs may switch between restartable and interruptible system call operation as often as desired in the execution of a program.

Issuing a **siginterrupt**(*3*) call during the execution of a signal handler will cause the new action to take place on the next signal to be caught.

NOTES

This library routine uses an extension of the sigaction(2) system call that is not available in 4.2BSD, hence it should not be used if backward compatibility is needed.

RETURN VALUES

A 0 value indicates that the call succeeded. A −1 value indicates that an invalid signal number has been supplied.

SEE ALSO

sigaction(2), sigprocmask(2), sigsuspend(2)

HISTORY

The **siginterrupt**() function appeared in 4.3BSD.

NAME

signal — simplified software signal facilities

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

void (*
signal(int sig, void (*func)(int)))(int);
```

DESCRIPTION

This **signal**() facility is a simplified interface to the more general sigaction(2) facility.

Signals allow the manipulation of a process from outside its domain as well as allowing the process to manipulate itself or copies of itself (children). There are two general types of signals: those that cause termination of a process and those that do not. Signals which cause termination of a program might result from an irrecoverable error or might be the result of a user at a terminal typing the 'interrupt' character. Signals are used when a process is stopped because it wishes to access its control terminal while in the background (see tty(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals result in the termination of the process receiving them if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the **signal**() function allows for a signal to be caught, to be ignored, or to generate an interrupt. See signal(7) for comprehensive list of supported signals.

The *func* procedure allows a user to choose the action upon receipt of a signal. To set the default action of the signal to occur as listed above, *func* should be SIG_DFL. A SIG_DFL resets the default action. To ignore the signal *func* should be SIG_IGN. This will cause subsequent instances of the signal to be ignored and pending instances to be discarded. If SIG_IGN is not used, further occurrences of the signal are automatically blocked and *func* is called.

The handled signal is unblocked when the function returns and the process continues from where it left off when the signal occurred. **Unlike previous signal facilities, the handler func() remains installed after a signal has been delivered.**

For some system calls, if a signal is caught while the call is executing and the call is prematurely terminated, the call is automatically restarted. (The handler is installed using the SA_RESTART flag with sigaction(2)). The affected system calls include read(2), write(2), sendto(2), recvfrom(2), sendmsg(2) and recvmsg(2) on a communications channel or a low speed device and during a ioctl(2) or wait(2). However, calls that have already committed are not restarted, but instead return a partial success (for example, a short read count).

When a process which has installed signal handlers forks, the child process inherits the signals. All caught signals may be reset to their default action by a call to the execve(2) function; ignored signals remain ignored.

Only functions that are async-signal-safe can safely be used in signal handlers, see signal(7) for a complete list.

RETURN VALUES

The previous action is returned on a successful call. Otherwise, SIG_ERR is returned and the global variable *errno* is set to indicate the error.

ERRORS

signal() will fail and no action will take place if one of the following occur:

[EINVAL] Specified *sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

SEE ALSO

kill(1), kill(2), ptrace(2), sigaction(2), sigaltstack(2), sigprocmask(2), sigsuspend(2), psignal(3), setjmp(3), strsignal(3), tty(4), signal(7)

HISTORY

This **signal**() facility appeared in 4.0BSD.

NAME

signbit — test sign

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <math.h>

int
signbit(real-floating x);
```

DESCRIPTION

The **signbit**() macro determines whether the sign of its argument value *x* is negative. An argument represented in a format wider than its semantic type is converted to its semantic type first. The determination is then based on the type of the argument.

IEEE754

The sign is determined for all values, including infinities, zeroes, and NaNs

VAX

The sign is determined for finites, true zeros, and dirty zeroes; for ROPs the sign is reported negative.

RETURN VALUES

The **signbit**() macro returns a non-zero value if the sign of its value *x* is negative. Otherwise 0 is returned.

ERRORS

No errors are defined.

SEE ALSO

fpclassify(3), isfinite(3), isnormal(3), math(3)

STANDARDS

The **signbit**() macro conforms to ISO/IEC 9899:1999 (“ISO C99”).

NAME

sigpause — atomically release blocked signals and wait for interrupt

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

int
sigpause(int sigmask);
```

DESCRIPTION

This interface is made obsolete by sigsuspend(**2**)**.**

sigpause() assigns *sigmask* to the set of masked signals and then waits for a signal to arrive; on return the set of masked signals is restored. *sigmask* is usually 0 to indicate that no signals are to be blocked. **sigpause**() always terminates by being interrupted, returning -1 with *errno* set to EINTR.

SEE ALSO

kill(2), sigaction(2), sigprocmask(2), sigsuspend(2), sigblock(3), sigvec(3)

HISTORY

The **sigpause**() function call appeared in 4.2BSD and has been deprecated.

NAME

sigrelse — manipulate current signal mask

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

int
sigrelse(int sig);
```

DESCRIPTION

This interface is made obsolete by `sigprocmask`**(2).**

The **sigrelse**() function removes the signal *sig* from the calling process' signal mask.

RETURN VALUES

If successful, the **sigrelse**() function returns 0. Otherwise −1 is returned and *errno* is set to indicate the error.

ERRORS

The **sigrelse**() function will fail if:

`[EINVAL]` The argument *sig* is not a valid signal number.

SEE ALSO

`sigprocmask`(2), `sighold`(3)

STANDARDS

The **sigrelse**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

sigset — manipulate signal dispositions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

void (*
sigset(int sig, void (*disp)(int)))(int);
```

DESCRIPTION

This interface is made obsolete by sigaction(**2**) **and** sigprocmask(**2**)**.**

The **sigset**() function manipulates the disposition of the signal *sig*. The new disposition is given in *disp*.

If *disp* is one of SIG_DFL, SIG_IGN, or the address of a handler function, the disposition of *sig* is changed accordingly, and *sig* is removed from the process' signal mask. Also, if *disp* is the address of a handler function, *sig* will be added to the process' signal mask during execution of the handler.

If *disp* is equal to SIG_HOLD, *sig* is added to the calling process' signal mask and the disposition of *sig* remains unchanged.

RETURN VALUES

If successful, the **sigset**() function returns SIG_HOLD if *sig* had been blocked, and the previous disposition of *sig* if it had not been blocked. Otherwise SIG_ERR is returned and *errno* is set to indicate the error.

ERRORS

The **sigset**() function will fail if:

[EINVAL] The argument *sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore a signal that cannot be ignored, such as SIGKILL or SIGSTOP.

SEE ALSO

sigaction(2), sigprocmask(2)

STANDARDS

The **sigset**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

sigsetmask — set current signal mask

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

int
sigsetmask(int mask);

sigmask(signum);
```

DESCRIPTION

This interface is made obsolete by: sigprocmask(2).

sigsetmask() sets the current signal mask Signals are blocked from delivery if the corresponding bit in *mask* is a 1; the macro **sigmask**() is provided to construct the mask for a given *signum*.

The system quietly disallows SIGKILL or SIGSTOP to be blocked.

RETURN VALUES

The previous set of masked signals is returned.

EXAMPLES

The following example using **sigsetmask**():

```
int omask;

omask = sigblock(sigmask(SIGINT) | sigmask(SIGHUP));

...

sigsetmask(omask & ~(sigmask(SIGINT) | sigmask(SIGHUP)));
```

Could be converted literally to:

```
sigset_t set, oset;

sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGHUP);
sigprocmask(SIG_BLOCK, &set, &oset);

...

sigdelset(&oset, SIGINT);
sigdelset(&oset, SIGHUP);
sigprocmask(SIG_SETMASK, &oset, NULL);
```

Another, clearer, alternative is:

```
sigset_t set;

sigemptyset(&set);
```

```
sigaddset(&set, SIGINT);
sigaddset(&set, SIGHUP);
sigprocmask(SIG_BLOCK, &set, NULL);

...

sigprocmask(SIG_UNBLOCK, &set, NULL);
```

To completely clear the signal mask using **sigsetmask**() one can do:

```
(void) sigsetmask(0);
```

Which can be expressed via sigprocmask(2) as:

```
sigset_t eset;

sigemptyset(&eset);
(void) sigprocmask(SIG_SETMASK, &eset, NULL);
```

SEE ALSO

kill(2), sigaction(2), sigprocmask(2), sigsuspend(2), sigblock(3), sigsetops(3), sigvec(3)

HISTORY

The **sigsetmask**() function call appeared in 4.2BSD and has been deprecated.

NAME

sigemptyset, **sigfillset**, **sigaddset**, **sigdelset**, **sigismember** — manipulate signal sets

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

int
sigemptyset(sigset_t *set);

int
sigfillset(sigset_t *set);

int
sigaddset(sigset_t *set, int signo);

int
sigdelset(sigset_t *set, int signo);

int
sigismember(sigset_t *set, int signo);
```

DESCRIPTION

These functions manipulate signal sets stored in a *sigset_t*. Either **sigemptyset**() or **sigfillset**() must be called for every object of type *sigset_t* before any other use of the object.

The **sigemptyset**() function initializes a signal set to be empty.

The **sigfillset**() function initializes a signal set to contain all signals.

The **sigaddset**() function adds the specified signal *signo* to the signal set.

The **sigdelset**() function deletes the specified signal *signo* from the signal set.

The **sigismember**() function returns whether a specified signal *signo* is contained in the signal set.

sigemptyset() and **sigfillset**() are provided as macros, but actual functions are available if their names are undefined (with #undef *name*).

RETURN VALUES

The **sigismember**() function returns 1 if the signal is a member of the set, a 0 otherwise. The other functions return 0 upon success. A −1 return value indicates an error occurred and the global variable *errno* is set to indicate the reason.

ERRORS

These functions could fail if one of the following occurs:

[EINVAL] *signo* has an invalid value.

SEE ALSO

kill(2), sigaction(2), sigsuspend(2), signal(7)

STANDARDS

These functions conform to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

sigvec — software signal facilities

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

struct sigvec {
        void    (*sv_handler)();
        int     sv_mask;
        int     sv_flags;
};

int
sigvec(int sig, struct sigvec *vec, struct sigvec *ovec);
```

DESCRIPTION

This interface is made obsolete by `sigaction`**(2). The structure, flags, and function declaration have been removed from the header files but the function is kept in the c library for binary compatibility.**

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs. A signal may also be *blocked*, in which case its delivery is postponed until it is *unblocked*. The action to be taken on delivery is determined at the time of delivery. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special *signal stack*.

Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a `sigblock`(3) or `sigsetmask`(3) call, or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it is delivered to the process. When a caught signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the process' signal handler (or until a `sigblock`(3) or `sigsetmask`(3) call is made). This mask is formed by taking the union of the current signal mask, the signal to be delivered, and the signal mask associated with the handler to be invoked.

sigvec() assigns a handler for a specific signal. If *vec* is non-zero, it specifies an action (`SIG_DFL`, `SIG_IGN`, or a handler routine) and mask to be used when delivering the specified signal. Further, if the `SV_ONSTACK` bit is set in *sv_flags*, the system will deliver the signal to the process on a *signal stack*, specified with `sigaltstack`(2). If *ovec* is non-zero, the previous handling information for the signal is returned to the user.

Once a signal handler is installed, it remains installed until another **sigvec**() call is made, or an execve(2) is performed. A signal-specific default action may be reset by setting *sv_handler* to SIG_DFL. The defaults are process termination, possibly with core dump; no action; stopping the process; or continuing the process. See the signal list below for each signal's default action. If *sv_handler* is set to SIG_DFL, the default action for the signal is to discard the signal, and if a signal is pending, the pending signal is discarded even if the signal is masked. If *sv_handler* is set to SIG_IGN, current and pending instances of the signal are ignored and discarded.

Options may be specified by setting *sv_flags*. If the SV_ONSTACK bit is set in *sv_flags*, the system will deliver the signal to the process on a *signal stack*, specified with sigstack(2).

If a signal is caught during the system calls listed below, the call may be restarted, the call may return with a data transfer shorter than requested, or the call may be forced to terminate with the error EINTR. Interrupting of pending calls is requested by setting the SV_INTERRUPT bit in *sv_flags*. The affected system calls include open(2), read(2), write(2), sendto(2), recvfrom(2), sendmsg(2) and recvmsg(2) on a communications channel or a slow device (such as a terminal, but not a regular file) and during a wait(2) or ioctl(2). However, calls that have already committed are not restarted, but instead return a partial success (for example, a short read count).

After a fork(2) or vfork(2) all signals, the signal mask, the signal stack, and the interrupt/restart flags are inherited by the child.

The execve(2) system call reinstates the default action for all signals which were caught and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; signals that interrupt pending system calls continue to do so.

See signal(7) for comprehensive list of supported signals.

NOTES

The mask specified in *vec* is not allowed to block SIGKILL or SIGSTOP. This is enforced silently by the system.

The SV_INTERRUPT flag is not available in 4.2BSD, hence it should not be used if backward compatibility is needed.

RETURN VALUES

A 0 value indicated that the call succeeded. A −1 return value indicates an error occurred and *errno* is set to indicated the reason.

EXAMPLES

The handler routine can be declared:

```
void
handler(sig, code, scp)
        int sig, code;
        struct sigcontext *scp;
```

Here *sig* is the signal number, into which the hardware faults and traps are mapped as defined below. *code* is a parameter that is either a constant or the code provided by the hardware. *scp* is a pointer to the *sigcontext* structure (defined in ⟨signal.h⟩), used to restore the context from before the signal.

ERRORS

sigvec() will fail and no new signal handler will be installed if one of the following occurs:

[EFAULT] Either *vec* or *ovec* points to memory that is not a valid part of the process address space.

[EINVAL] *sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

SEE ALSO

kill(1), kill(2), ptrace(2), sigaction(2), sigaltstack(2), sigprocmask(2), sigstack(2), sigsuspend(2), setjmp(3), sigblock(3), siginterrupt(3), signal(3), sigpause(3), sigsetmask(3), sigsetops(3), tty(4), signal(7)

NAME

sin, **sinf** — sine function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
sin(double x);

float
sinf(float x);
```

DESCRIPTION

The **sin**() function computes the sine of *x* (measured in radians). A large magnitude argument may yield a result with little or no significance.

RETURN VALUES

The **sin**() function returns the sine value.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **sin**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

sinh, **sinhf** — hyperbolic sine function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
sinh(double x);

float
sinhf(float x);
```

DESCRIPTION

The **sinh**() function computes the hyperbolic sine of *x*.

RETURN VALUES

The **sinh**() function returns the hyperbolic sine value unless the magnitude of *x* is too large; in this event, the global variable *errno* is set to ERANGE.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sin(3), tan(3), tanh(3)

STANDARDS

The **sinh**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

skey, **skeychallenge**, **skeylookup**, **skeygetnext**, **skeyverify**, **skeyzero**, **getskeyprompt**, **skey_set_algorithm**, **skey_get_algorithm**, **skey_haskey**, **skey_keyinfo**, **skey_passcheck**, **skey_authenticate** — one-time password (OTP) library

LIBRARY

S/key One-Time Password Library (libskey, -lskey)

SYNOPSIS

```
#include <skey.h>

int
skeychallenge(struct skey *mp, const char *name, char *ss, size_t sslen);

int
skeylookup(struct skey *mp, const char *name);

int
skeygetnext(struct skey *mp);

int
skeyverify(struct skey *mp, char *response);

int
skeyzero(struct skey *mp, char *response);

int
getskeyprompt(struct skey *mp, char *name, char *prompt);

const char *
skey_set_algorithm(const char *new);

const char *
skey_get_algorithm(void);

int
skey_haskey(const char *username);

const char *
skey_keyinfo(const char *username);

int
skey_passcheck(const char *username, char *passwd);

int
skey_authenticate(const char *username);

void
f(char *x);

int
keycrunch(char *result, const char *seed, const char *passwd);

void
rip(char *buf);

char *
readpass(char *buf, int n);
```

*char **
readskey(*char *buf*, *int n*);

int
atob8(*char *out*, *const char *in*);

int
btoa8(*char *out*, *const char *in*);

int
htoi(*int c*);

*const char **
skipspace(*const char *cp*);

void
backspace(*char *buf*);

void
sevenbit(*char *buf*);

*char **
btoe(*char *engout*, *const char *c*);

int
etob(*char *out*, *const char *e*);

*char **
put8(*char *out*, *const char *s*);

DESCRIPTION

The **skey** library provides routines for accessing NetBSD's one-time password (OTP) authentication system.

Most S/Key operations take a pointer to a *struct skey*, which should be considered as an opaque identifier.

FUNCTIONS

The following high-level functions are available:

skeychallenge(*mp*, *name*, *ss*, *sslen*)
: Return a S/Key challenge for user *name*. If successful, the caller's skey structure *mp* is filled and 0 is returned. If unsuccessful (e.g. if name is unknown), −1 is returned.

skeylookup(*mp*, *name*)
: Find an entry for user *name* in the one-time password database. Returns 0 if the entry is found and 1 if the entry is not found. If an error occurs accessing the database, −1 is returned.

skeygetnext(*mp*)
: Get the next entry in the one-time password database. Returns 0 on success and the entry is stored in *mp* and 1 if no more entries are available. If an error occurs accessing the database, −1 is returned.

skeyverify(*mp*, *response*)
: Verify response *response* to a S/Key challenge. Returns 0 if the verification is successful and 1 if the verification failed. If an error occurs accessing the database, −1 is returned.

skeyzero(*mp*, *response*)
: Comment out user's entry in the S/Key database. Returns 0 on success and the database is updated, otherwise −1 is returned and the database remains unchanged.

getskeyprompt(*mp*, *name*, *prompt*)
Issue a S/Key challenge for user *name*. If successful, fill in the caller's skey structure *mp* and return 0. If unsuccessful (e.g. if name is unknown) −1 is returned.

The following lower-level functions are available:

skey_set_algorithm(*new*)
Set hash algorithm type. Valid values for *new* are "md4", "md5" and "sha1".

skey_get_algorithm(*void*)
Get current hash type.

skey_haskey(*username*)
Returns 0 if the user *username* exists and 1 if the user doesn't exist. Returns −1 on file error.

skey_keyinfo(*username*)
Returns the current sequence number and seed for user *username*.

skey_passcheck(*username*, *passwd*)
Checks to see if answer is the correct one to the current challenge.

skey_authenticate(*username*)
Used when calling program will allow input of the user's response to the challenge. Returns zero on success or −1 on failure.

The following miscellaneous functions are available:

f(*x*) One-way function to take 8 bytes pointed to by *x* and return 8 bytes in place.

keycrunch(*char *result*, *const char *seed*, *const char *passwd*)
Crunch a key.

rip(*buf*)
Strip trailing CR/LF characters from a line of text *buf*.

readpass(*buf*, *n*)
Read in secret passwd (turns off echo).

readskey(*buf*, *n*)
Read in an s/key OTP (does not turn off echo).

atob8(*out*, *in*)
Convert 8-byte hex-ascii string *in* to binary array *out*. Returns 0 on success, −1 on error.

btoa8(*out*, *in*)
Convert 8-byte binary array *in* to hex-ascii string *out*. Returns 0 on success, −1 on error.

htoi(*int c*)
Convert hex digit to binary integer.

skipspace(*cp*)
Skip leading spaces from the string *cp*.

backspace(*buf*)
Remove backspaced over characters from the string *buf*.

sevenbit(*buf*)
Ensure line *buf* is all seven bits.

btoe(*engout*, *c*)
Encode 8 bytes in *c* as a string of English words. Returns a pointer to a static buffer in *engout*.

etob(*out*, *e*)
Convert English to binary. Returns 0 if the word is not in the database, 1 if all good words and parity is valid, −1 if badly formed input (i.e. > 4 char word) and -2 if words are valid but parity is wrong.

put8(*out*, *s*)
Display 8 bytes *s* as a series of 16-bit hex digits.

FILES

`/usr/lib/libskey.a`	static skey library
`/usr/lib/libskey.so`	dynamic skey library
`/usr/lib/libskey_p.a`	static skey library compiled for profiling

SEE ALSO

`skey`(1), `skeyaudit`(1), `skeyinfo`(1)

BUGS

The **skey** library functions are not re-entrant or thread-safe.

The **skey** library defines many poorly named functions which pollute the name space.

NAME

sleep — suspend process execution for interval of seconds

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

unsigned int
sleep(unsigned int seconds);
```

DESCRIPTION

The **sleep**() function suspends execution of the calling process until either the number of seconds specified by *seconds* have elapsed or a signal is delivered to the calling process and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system.

RETURN VALUES

If the **sleep**() function returns because the requested time has elapsed, the value returned will be zero. If the **sleep**() function returns due to the delivery of a signal, the value returned will be the unslept amount (the request time minus the time actually slept) in seconds.

SEE ALSO

nanosleep(2), usleep(3)

STANDARDS

The **sleep**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

A **sleep**() function appeared in Version 7 AT&T UNIX.

NAME

snprintb — bitmask output conversion

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
snprintb(char *buf, size_t buflen, const char *fmt, uint64_t val);
```

DESCRIPTION

The **snprintb**() function formats a bitmask into a mnemonic form suitable for printing.

This conversion is useful for decoding bit fields in device registers. It formats the integer *val* into the buffer *buf*, of size *buflen*, using a specified radix and an interpretation of the bits within that integer as though they were flags.

The decoding directive string *fmt* describes how the bitfield is to be interpreted and displayed. The first character of *fmt* is a binary character representation of the output numeral base in which the bitfield will be printed before it is decoded. Recognized radix values (in C escape-character format) are `\10` (octal), `\12` (decimal), and `\20` (hexadecimal).

The remaining characters in *fmt* are interpreted as a list of bit-position–description pairs. A bit-position–description pair begins with a binary character value that represents the position of the bit being described. A bit position value of one describes the least significant bit. Whereas a position value of 32 (octal 40, hexadecimal 20, the ASCII space character) describes the most significant bit.

The remaining characters in a bit-position–description pair are the characters to print should the bit being described be set. Description strings are delimited by the next bit position value character encountered (distinguishable by its value being ≤ 32), or the end of the decoding directive string itself.

RETURN VALUES

The **snprintb**() function returns the number of characters that are required to format the value *val* given the format string *fmt* excluding the terminating NUL. The returned string in *buf* is always NUL-terminated.

EXAMPLES

Two examples of the old formatting style:

```
snprintb(buf, buflen, "\10\2BITTWO\1BITONE", 3)
⇒ "3<BITTWO,BITONE>"

snprintb(buf, buflen
       "\20\x10NOTBOOT\x0fFPP\x0eSDVMA\x0cVIDEO"
       "\x0bLORES\x0aFPA\x09DIAG\x07CACHE"
       "\x06IOCACHE\x05LOOPBACK\x04DBGCACHE",
       0xe860)
⇒ "0xe860<NOTBOOT,FPP,SDVMA,VIDEO,CACHE,IOCACHE>"
```

ERRORS

If the buffer *buf* is too small to hold the formatted output, **snprintb**() will still return the buffer, containing a truncated string.

SEE ALSO

`snprintf`(3)

HISTORY

The **`snprintb`**() function was originally implemented as a non-standard `%b` format string for the kernel **`printf`**() function in NetBSD 1.5 and earlier releases. It got implemented as **`bitmap_snprintf`**() for NetBSD 1.6 and this version was used to implement **`snprintb`**().

BUGS

`snprintb`() supports a new extended form of formatting string, which is not yet described here.

NAME

sockaddr_snprintf — formatting function for socket address structures

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
sockaddr_snprintf(char *buf, size_t buflen, const char *fmt,
    const struct sockaddr *sa);
```

DESCRIPTION

The **sockaddr_snprintf**() function formats a socket address into a form suitable for printing.

This function is convenient because it is protocol independent, i.e. one does not need to know the address family of the sockaddr in order to print it. The printf(3) like format string specifies how the address is going to be printed. Some formatting characters are only supported by some address families. If a certain formatting character is not supported, then the string "N/A" is printed.

The resulting formatted string is placed into *buf*. Up to *buflen* characters are placed in *buf*.

The following formatting characters are supported (immediately after a percent ('%') sign):

a The node address portion of the socket address is printed numerically. For AF_INET the address is printed as: "A.B.C.D" and for AF_INET6 the address is printed as: "A:B...[%if]" using getnameinfo(3) internally with NI_NUMERICHOST. For AF_APPLETALK the address is printed as: "A.B" For AF_LOCAL (AF_UNIX) the address is printed as: "socket-path" For AF_LINK the address is printed as: "a.b.c.d.e.f" using link_ntoa(3), but the interface portion is skipped (see below). For AF_UNSPEC nothing is printed.

A The symbolic name of the address is printed. For AF_INET and this is the hostname associated with the address. For all other address families, it is the same as the "a" format.

f The numeric value of the family of the address is printed.

l The length of the socket address is printed.

p For AF_INET, AF_INET6, and AF_APPLETALK the numeric value of the port portion of the address is printed.

P For AF_INET and AF_INET6 this is the name of the service associated with the port number, if available. For all other address families, it is the same as the "p" format.

I For AF_LINK addresses, the interface name portion is printed.

F For AF_INET6 addresses, the flowinfo portion of the address is printed numerically.

R For AF_APPLETALK addresses, the netrange portion of the address is printed as: "phase:[firstnet,lastnet]"

S For AF_INET6 addresses, the scope portion of the address is printed numerically.

? If present between "%" and the format character, and the selected format does not apply to the given address family, the "N/A" string is elided and no output results.

RETURN VALUES

The **sockaddr_snprintf**() function returns the number of characters that are required to format the value *val* given the format string *fmt* excluding the terminating NUL. The returned string in *buf* is always NUL-terminated. If the address family is not supported, **sockaddr_snprintf**() returns −1 and sets *errno* to EAFNOSUPPORT. For AF_INET and addresses **sockaddr_snprintf**() returns −1 if the getnameinfo(3) conversion failed, and *errno* is set to the error value from getnameinfo(3).

ERRORS

If the buffer *buf* is too small to hold the formatted output, **sockaddr_snprintf**() will still return the buffer, containing a truncated string.

SEE ALSO

getaddrinfo(3), getnameinfo(3), link_ntoa(3), snprintf(3)

HISTORY

The **sockaddr_snprintf**() first appeared in NetBSD 3.0.

BUGS

The **sockaddr_snprintf**() interface is experimental and might change in the future.

There is no way to specify different formatting styles for particular addresses. For example it would be useful to print AF_LINK addresses as “%.2x:%.2x...” instead of “%x.%x...”

This function is supposed to be quick, but getnameinfo(3) might use system calls to convert the scope number to an interface name and the “A” and “P” format characters call getaddrinfo(3) which may block for a noticeable period of time.

Not all formatting characters are supported by all address families and printing “N/A” is not very convenient. The “?” character can suppress this, but other formatting (e.g., spacing or punctuation) will remain.

NAME

sockatmark — determine whether a socket is at the out-of-band mark

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/socket.h>

int
sockatmark(int s);
```

DESCRIPTION

The **sockatmark** function determines whether the socket referenced by the file descriptor *s* is at the out-of-band mark.

RETURN VALUES

If successful, the **sockatmark** function returns 1 to indicate that the socket is at an out-of-band mark; 0 is returned if there is no out-of-band mark or the mark is preceded by in-band data. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The **sockatmark** function will fail if:

[EBADF] The argument *s* is not a valid file descriptor.

[ENOTTY] The file descriptor *s* does not refer to a socket.

SEE ALSO

ioctl(2), recv(2), socket(2)

Stuart Sechrest, *An Introductory 4.4BSD Interprocess Communication Tutorial*. (see /usr/share/doc/psd/20.ipctut)

Samuel J. Leffler, Robert S. Fabry, William N. Joy, Phil Lapsley, Steve Miller, and Chris Torek, *Advanced 4.4BSD IPC Tutorial*. (see /usr/share/doc/psd/21.ipc)

STANDARDS

The **sockatmark** function conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

HISTORY

The **sockatmark** function appeared in IEEE Std 1003.1g-2000 (“POSIX.1”) as a replacement for the SIOCATMARK ioctl(2) interface.

NAME

cbrt, **cbrtf**, **sqrt**, **sqrtf** — cube root and square root functions

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
cbrt(double x);

float
cbrtf(float x);

double
sqrt(double x);

float
sqrtf(float x);
```

DESCRIPTION

The **cbrt**() and **cbrtf**() functions compute the cube root of *x*.

The **sqrt**() and **sqrtf**() functions compute the non-negative square root of x.

RETURN VALUES

If x is negative, **sqrt**(*x*) and **sqrtf**(*x*) set the global variable *errno* to EDOM.

SEE ALSO

math(3)

STANDARDS

The **sqrt**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

HISTORY

The **cbrt**() function appeared in 4.3BSD.

NAME

SSL − OpenSSL SSL/TLS library

LIBRARY

libcrypto, -lcrypto

SYNOPSIS

DESCRIPTION

The OpenSSL **ssl** library implements the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols. It provides a rich API which is documented here.

At first the library must be initialized; see *SSL_library_init*(3).

Then an **SSL_CTX** object is created as a framework to establish TLS/SSL enabled connections (see *SSL_CTX_new*(3)). Various options regarding certificates, algorithms etc. can be set in this object.

When a network connection has been created, it can be assigned to an **SSL** object. After the **SSL** object has been created using *SSL_new*(3), *SSL_set_fd*(3) or *SSL_set_bio*(3) can be used to associate the network connection with the object.

Then the TLS/SSL handshake is performed using *SSL_accept*(3) or *SSL_connect*(3) respectively. *SSL_read*(3) and *SSL_write*(3) are used to read and write data on the TLS/SSL connection. *SSL_shutdown*(3) can be used to shut down the TLS/SSL connection.

DATA STRUCTURES

Currently the OpenSSL **ssl** library functions deals with the following data structures:

SSL_METHOD (SSL Method)

That's a dispatch structure describing the internal **ssl** library methods/functions which implement the various protocol versions (SSLv1, SSLv2 and TLSv1). It's needed to create an **SSL_CTX**.

SSL_CIPHER (SSL Cipher)

This structure holds the algorithm information for a particular cipher which are a core part of the SSL/TLS protocol. The available ciphers are configured on a **SSL_CTX** basis and the actually used ones are then part of the **SSL_SESSION**.

SSL_CTX (SSL Context)

That's the global context structure which is created by a server or client once per program life-time and which holds mainly default values for the **SSL** structures which are later created for the connections.

SSL_SESSION (SSL Session)

This is a structure containing the current TLS/SSL session details for a connection: **SSL_CIPHER**s, client and server certificates, keys, etc.

SSL (SSL Connection)

That's the main SSL/TLS structure which is created by a server or client per established connection. This actually is the core structure in the SSL API. Under run-time the application usually deals with this structure which has links to mostly all other structures.

HEADER FILES

Currently the OpenSSL **ssl** library provides the following C header files containing the prototypes for the data structures and and functions:

ssl.h

That's the common header file for the SSL/TLS API. Include it into your program to make the API of the **ssl** library available. It internally includes both more private SSL headers and headers from the **crypto** library. Whenever you need hard-core details on the internals of the SSL API, look inside this header file.

ssl2.h

That's the sub header file dealing with the SSLv2 protocol only. *Usually you don't have to include it explicitly because it's already included by ssl.h.*

ssl3.h
That's the sub header file dealing with the SSLv3 protocol only. *Usually you don't have to include it explicitly because it's already included by ssl.h.*

ssl23.h
That's the sub header file dealing with the combined use of the SSLv2 and SSLv3 protocols. *Usually you don't have to include it explicitly because it's already included by ssl.h.*

tls1.h
That's the sub header file dealing with the TLSv1 protocol only. *Usually you don't have to include it explicitly because it's already included by ssl.h.*

API FUNCTIONS

Currently the OpenSSL **ssl** library exports 214 API functions. They are documented in the following:

DEALING WITH PROTOCOL METHODS

Here we document the various API functions which deal with the SSL/TLS protocol methods defined in **SSL_METHOD** structures.

const SSL_METHOD ***SSLv2_client_method**(void);
Constructor for the SSLv2 SSL_METHOD structure for a dedicated client.

const SSL_METHOD ***SSLv2_server_method**(void);
Constructor for the SSLv2 SSL_METHOD structure for a dedicated server.

const SSL_METHOD ***SSLv2_method**(void);
Constructor for the SSLv2 SSL_METHOD structure for combined client and server.

const SSL_METHOD ***SSLv3_client_method**(void);
Constructor for the SSLv3 SSL_METHOD structure for a dedicated client.

const SSL_METHOD ***SSLv3_server_method**(void);
Constructor for the SSLv3 SSL_METHOD structure for a dedicated server.

const SSL_METHOD ***SSLv3_method**(void);
Constructor for the SSLv3 SSL_METHOD structure for combined client and server.

const SSL_METHOD ***TLSv1_client_method**(void);
Constructor for the TLSv1 SSL_METHOD structure for a dedicated client.

const SSL_METHOD ***TLSv1_server_method**(void);
Constructor for the TLSv1 SSL_METHOD structure for a dedicated server.

const SSL_METHOD ***TLSv1_method**(void);
Constructor for the TLSv1 SSL_METHOD structure for combined client and server.

DEALING WITH CIPHERS

Here we document the various API functions which deal with the SSL/TLS ciphers defined in **SSL_CIPHER** structures.

char ***SSL_CIPHER_description**(SSL_CIPHER *cipher, char *buf, int len);
Write a string to *buf* (with a maximum size of *len*) containing a human readable description of *cipher*. Returns *buf*.

int **SSL_CIPHER_get_bits**(SSL_CIPHER *cipher, int *alg_bits);
Determine the number of bits in *cipher*. Because of export crippled ciphers there are two bits: The bits the algorithm supports in general (stored to *alg_bits*) and the bits which are actually used (the return value).

const char ***SSL_CIPHER_get_name**(SSL_CIPHER *cipher);
Return the internal name of *cipher* as a string. These are the various strings defined by the *SSL2_TXT_xxx*, *SSL3_TXT_xxx* and *TLS1_TXT_xxx* definitions in the header files.

char *__SSL_CIPHER_get_version__(SSL_CIPHER *cipher);

Returns a string like "`TLSv1/SSLv3`" or "`SSLv2`" which indicates the SSL/TLS protocol version to which *cipher* belongs (i.e. where it was defined in the specification the first time).

DEALING WITH PROTOCOL CONTEXTS

Here we document the various API functions which deal with the SSL/TLS protocol context defined in the **SSL_CTX** structure.

int **SSL_CTX_add_client_CA**(SSL_CTX *ctx, X509 *x);
long **SSL_CTX_add_extra_chain_cert**(SSL_CTX *ctx, X509 *x509);
int **SSL_CTX_add_session**(SSL_CTX *ctx, SSL_SESSION *c);
int **SSL_CTX_check_private_key**(const SSL_CTX *ctx);
long **SSL_CTX_ctrl**(SSL_CTX *ctx, int cmd, long larg, char *parg);
void **SSL_CTX_flush_sessions**(SSL_CTX *s, long t);
void **SSL_CTX_free**(SSL_CTX *a);
char *__SSL_CTX_get_app_data__(SSL_CTX *ctx);
X509_STORE *__SSL_CTX_get_cert_store__(SSL_CTX *ctx);
STACK *__SSL_CTX_get_client_CA_list__(const SSL_CTX *ctx);
int (*__SSL_CTX_get_client_cert_cb__(SSL_CTX *ctx))(SSL *ssl, X509 **x509, EVP_PKEY **pkey);
char *__SSL_CTX_get_ex_data__(const SSL_CTX *s, int idx);
int **SSL_CTX_get_ex_new_index**(long argl, char *argp, int (*new_func);(void), int (*dup_func)(void), void (*free_func)(void))
void (*__SSL_CTX_get_info_callback__(SSL_CTX *ctx))(SSL *ssl, int cb, int ret);
int **SSL_CTX_get_quiet_shutdown**(const SSL_CTX *ctx);
int **SSL_CTX_get_session_cache_mode**(SSL_CTX *ctx);
long **SSL_CTX_get_timeout**(const SSL_CTX *ctx);
int (*__SSL_CTX_get_verify_callback__(const SSL_CTX *ctx))(int ok, X509_STORE_CTX *ctx);
int **SSL_CTX_get_verify_mode**(SSL_CTX *ctx);
int **SSL_CTX_load_verify_locations**(SSL_CTX *ctx, char *CAfile, char *CApath);
long **SSL_CTX_need_tmp_RSA**(SSL_CTX *ctx);
SSL_CTX *__SSL_CTX_new__(const SSL_METHOD *meth);
int **SSL_CTX_remove_session**(SSL_CTX *ctx, SSL_SESSION *c);
int **SSL_CTX_sess_accept**(SSL_CTX *ctx);
int **SSL_CTX_sess_accept_good**(SSL_CTX *ctx);
int **SSL_CTX_sess_accept_renegotiate**(SSL_CTX *ctx);
int **SSL_CTX_sess_cache_full**(SSL_CTX *ctx);
int **SSL_CTX_sess_cb_hits**(SSL_CTX *ctx);
int **SSL_CTX_sess_connect**(SSL_CTX *ctx);
int **SSL_CTX_sess_connect_good**(SSL_CTX *ctx);
int **SSL_CTX_sess_connect_renegotiate**(SSL_CTX *ctx);
int **SSL_CTX_sess_get_cache_size**(SSL_CTX *ctx);
SSL_SESSION *(*__SSL_CTX_sess_get_get_cb__(SSL_CTX *ctx))(SSL *ssl, unsigned char *data, int len, int *copy);
int (*__SSL_CTX_sess_get_new_cb__(SSL_CTX *ctx)(SSL *ssl, SSL_SESSION *sess);
void (*__SSL_CTX_sess_get_remove_cb__(SSL_CTX *ctx)(SSL_CTX *ctx, SSL_SESSION *sess);
int **SSL_CTX_sess_hits**(SSL_CTX *ctx);
int **SSL_CTX_sess_misses**(SSL_CTX *ctx);
int **SSL_CTX_sess_number**(SSL_CTX *ctx);
void **SSL_CTX_sess_set_cache_size**(SSL_CTX *ctx,t);
void **SSL_CTX_sess_set_get_cb**(SSL_CTX *ctx, SSL_SESSION *(*cb)(SSL *ssl, unsigned char *data, int len, int *copy));
void **SSL_CTX_sess_set_new_cb**(SSL_CTX *ctx, int (*cb)(SSL *ssl, SSL_SESSION *sess));
void **SSL_CTX_sess_set_remove_cb**(SSL_CTX *ctx, void (*cb)(SSL_CTX *ctx, SSL_SESSION *sess));

int **SSL_CTX_sess_timeouts**(SSL_CTX *ctx);
LHASH ***SSL_CTX_sessions**(SSL_CTX *ctx);
void **SSL_CTX_set_app_data**(SSL_CTX *ctx, void *arg);
void **SSL_CTX_set_cert_store**(SSL_CTX *ctx, X509_STORE *cs);
void **SSL_CTX_set_cert_verify_cb**(SSL_CTX *ctx, int (*cb)(), char *arg)
int **SSL_CTX_set_cipher_list**(SSL_CTX *ctx, char *str);
void **SSL_CTX_set_client_CA_list**(SSL_CTX *ctx, STACK *list);
void **SSL_CTX_set_client_cert_cb**(SSL_CTX *ctx, int (*cb)(SSL *ssl, X509 **x509, EVP_PKEY **pkey));
void **SSL_CTX_set_default_passwd_cb**(SSL_CTX *ctx, int (*cb);(void))
void **SSL_CTX_set_default_read_ahead**(SSL_CTX *ctx, int m);
int **SSL_CTX_set_default_verify_paths**(SSL_CTX *ctx);
int **SSL_CTX_set_ex_data**(SSL_CTX *s, int idx, char *arg);
void **SSL_CTX_set_info_callback**(SSL_CTX *ctx, void (*cb)(SSL *ssl, int cb, int ret));
void **SSL_CTX_set_msg_callback**(SSL_CTX *ctx, void (*cb)(int write_p, int version, int content_type, const void *buf, size_t len, SSL *ssl, void *arg));
void **SSL_CTX_set_msg_callback_arg**(SSL_CTX *ctx, void *arg);
void **SSL_CTX_set_options**(SSL_CTX *ctx, unsigned long op);
void **SSL_CTX_set_quiet_shutdown**(SSL_CTX *ctx, int mode);
void **SSL_CTX_set_session_cache_mode**(SSL_CTX *ctx, int mode);
int **SSL_CTX_set_ssl_version**(SSL_CTX *ctx, const SSL_METHOD *meth);
void **SSL_CTX_set_timeout**(SSL_CTX *ctx, long t);
long **SSL_CTX_set_tmp_dh**(SSL_CTX* ctx, DH *dh);
long **SSL_CTX_set_tmp_dh_callback**(SSL_CTX *ctx, DH *(*cb)(void));
long **SSL_CTX_set_tmp_rsa**(SSL_CTX *ctx, RSA *rsa);
SSL_CTX_set_tmp_rsa_callback

```
long SSL_CTX_set_tmp_rsa_callback(SSL_CTX *ctx, RSA *(*cb)(SSL *ssl,
int export, int keylength));
```

Sets the callback which will be called when a temporary private key is required. The **export** flag will be set if the reason for needing a temp key is that an export ciphersuite is in use, in which case, **keylength** will contain the required keylength in bits. Generate a key of appropriate size (using ???) and return it.

SSL_set_tmp_rsa_callback

long **SSL_set_tmp_rsa_callback**(SSL *ssl, RSA *(*cb)(SSL *ssl, int export, int keylength));

The same as **SSL_CTX_set_tmp_rsa_callback**, except it operates on an SSL session instead of a context.

void **SSL_CTX_set_verify**(SSL_CTX *ctx, int mode, int (*cb);(void))
int **SSL_CTX_use_PrivateKey**(SSL_CTX *ctx, EVP_PKEY *pkey);
int **SSL_CTX_use_PrivateKey_ASN1**(int type, SSL_CTX *ctx, unsigned char *d, long len);
int **SSL_CTX_use_PrivateKey_file**(SSL_CTX *ctx, char *file, int type);
int **SSL_CTX_use_RSAPrivateKey**(SSL_CTX *ctx, RSA *rsa);
int **SSL_CTX_use_RSAPrivateKey_ASN1**(SSL_CTX *ctx, unsigned char *d, long len);
int **SSL_CTX_use_RSAPrivateKey_file**(SSL_CTX *ctx, char *file, int type);
int **SSL_CTX_use_certificate**(SSL_CTX *ctx, X509 *x);
int **SSL_CTX_use_certificate_ASN1**(SSL_CTX *ctx, int len, unsigned char *d);
int **SSL_CTX_use_certificate_file**(SSL_CTX *ctx, char *file, int type);
void **SSL_CTX_set_psk_client_callback**(SSL_CTX *ctx, unsigned int (*callback)(SSL *ssl, const char *hint, char *identity, unsigned int max_identity_len, unsigned char *psk, unsigned int max_psk_len));
int **SSL_CTX_use_psk_identity_hint**(SSL_CTX *ctx, const char *hint);
void **SSL_CTX_set_psk_server_callback**(SSL_CTX *ctx, unsigned int (*callback)(SSL *ssl, const char *identity, unsigned char *psk, int max_psk_len));

DEALING WITH SESSIONS

Here we document the various API functions which deal with the SSL/TLS sessions defined in the **SSL_SESSION** structures.

int **SSL_SESSION_cmp**(const SSL_SESSION *a, const SSL_SESSION *b);
void **SSL_SESSION_free**(SSL_SESSION *ss);
char ***SSL_SESSION_get_app_data**(SSL_SESSION *s);
char ***SSL_SESSION_get_ex_data**(const SSL_SESSION *s, int idx);
int **SSL_SESSION_get_ex_new_index**(long argl, char *argp, int (*new_func);(void), int (*dup_func)(void), void (*free_func)(void))
long **SSL_SESSION_get_time**(const SSL_SESSION *s);
long **SSL_SESSION_get_timeout**(const SSL_SESSION *s);
unsigned long **SSL_SESSION_hash**(const SSL_SESSION *a);
SSL_SESSION ***SSL_SESSION_new**(void);
int **SSL_SESSION_print**(BIO *bp, const SSL_SESSION *x);
int **SSL_SESSION_print_fp**(FILE *fp, const SSL_SESSION *x);
void **SSL_SESSION_set_app_data**(SSL_SESSION *s, char *a);
int **SSL_SESSION_set_ex_data**(SSL_SESSION *s, int idx, char *arg);
long **SSL_SESSION_set_time**(SSL_SESSION *s, long t);
long **SSL_SESSION_set_timeout**(SSL_SESSION *s, long t);

DEALING WITH CONNECTIONS

Here we document the various API functions which deal with the SSL/TLS connection defined in the **SSL** structure.

int **SSL_accept**(SSL *ssl);
int **SSL_add_dir_cert_subjects_to_stack**(STACK *stack, const char *dir);
int **SSL_add_file_cert_subjects_to_stack**(STACK *stack, const char *file);
int **SSL_add_client_CA**(SSL *ssl, X509 *x);
char ***SSL_alert_desc_string**(int value);
char ***SSL_alert_desc_string_long**(int value);
char ***SSL_alert_type_string**(int value);
char ***SSL_alert_type_string_long**(int value);
int **SSL_check_private_key**(const SSL *ssl);
void **SSL_clear**(SSL *ssl);
long **SSL_clear_num_renegotiations**(SSL *ssl);
int **SSL_connect**(SSL *ssl);
void **SSL_copy_session_id**(SSL *t, const SSL *f);
long **SSL_ctrl**(SSL *ssl, int cmd, long larg, char *parg);
int **SSL_do_handshake**(SSL *ssl);
SSL ***SSL_dup**(SSL *ssl);
STACK ***SSL_dup_CA_list**(STACK *sk);
void **SSL_free**(SSL *ssl);
SSL_CTX ***SSL_get_SSL_CTX**(const SSL *ssl);
char ***SSL_get_app_data**(SSL *ssl);
X509 ***SSL_get_certificate**(const SSL *ssl);
const char ***SSL_get_cipher**(const SSL *ssl);
int **SSL_get_cipher_bits**(const SSL *ssl, int *alg_bits);
char ***SSL_get_cipher_list**(const SSL *ssl, int n);
char ***SSL_get_cipher_name**(const SSL *ssl);
char ***SSL_get_cipher_version**(const SSL *ssl);
STACK ***SSL_get_ciphers**(const SSL *ssl);
STACK ***SSL_get_client_CA_list**(const SSL *ssl);

SSL_CIPHER *__SSL_get_current_cipher__(SSL *ssl);
long __SSL_get_default_timeout__(const SSL *ssl);
int __SSL_get_error__(const SSL *ssl, int i);
char *__SSL_get_ex_data__(const SSL *ssl, int idx);
int __SSL_get_ex_data_X509_STORE_CTX_idx__(void);
int __SSL_get_ex_new_index__(long argl, char *argp, int (*new_func);(void), int (*dup_func)(void), void (*free_func)(void))
int __SSL_get_fd__(const SSL *ssl);
void (*__SSL_get_info_callback__(const SSL *ssl);)()
STACK *__SSL_get_peer_cert_chain__(const SSL *ssl);
X509 *__SSL_get_peer_certificate__(const SSL *ssl);
EVP_PKEY *__SSL_get_privatekey__(SSL *ssl);
int __SSL_get_quiet_shutdown__(const SSL *ssl);
BIO *__SSL_get_rbio__(const SSL *ssl);
int __SSL_get_read_ahead__(const SSL *ssl);
SSL_SESSION *__SSL_get_session__(const SSL *ssl);
char *__SSL_get_shared_ciphers__(const SSL *ssl, char *buf, int len);
int __SSL_get_shutdown__(const SSL *ssl);
const SSL_METHOD *__SSL_get_ssl_method__(SSL *ssl);
int __SSL_get_state__(const SSL *ssl);
long __SSL_get_time__(const SSL *ssl);
long __SSL_get_timeout__(const SSL *ssl);
int (*__SSL_get_verify_callback__(const SSL *ssl))(int,X509_STORE_CTX *)
int __SSL_get_verify_mode__(const SSL *ssl);
long __SSL_get_verify_result__(const SSL *ssl);
char *__SSL_get_version__(const SSL *ssl);
BIO *__SSL_get_wbio__(const SSL *ssl);
int __SSL_in_accept_init__(SSL *ssl);
int __SSL_in_before__(SSL *ssl);
int __SSL_in_connect_init__(SSL *ssl);
int __SSL_in_init__(SSL *ssl);
int __SSL_is_init_finished__(SSL *ssl);
STACK *__SSL_load_client_CA_file__(char *file);
void __SSL_load_error_strings__(void);
SSL *__SSL_new__(SSL_CTX *ctx);
long __SSL_num_renegotiations__(SSL *ssl);
int __SSL_peek__(SSL *ssl, void *buf, int num);
int __SSL_pending__(const SSL *ssl);
int __SSL_read__(SSL *ssl, void *buf, int num);
int __SSL_renegotiate__(SSL *ssl);
char *__SSL_rstate_string__(SSL *ssl);
char *__SSL_rstate_string_long__(SSL *ssl);
long __SSL_session_reused__(SSL *ssl);
void __SSL_set_accept_state__(SSL *ssl);
void __SSL_set_app_data__(SSL *ssl, char *arg);
void __SSL_set_bio__(SSL *ssl, BIO *rbio, BIO *wbio);
int __SSL_set_cipher_list__(SSL *ssl, char *str);
void __SSL_set_client_CA_list__(SSL *ssl, STACK *list);
void __SSL_set_connect_state__(SSL *ssl);
int __SSL_set_ex_data__(SSL *ssl, int idx, char *arg);
int __SSL_set_fd__(SSL *ssl, int fd);
void __SSL_set_info_callback__(SSL *ssl, void (*cb);(void))

void **SSL_set_msg_callback**(SSL *ctx, void (*cb)(int write_p, int version, int content_type, const void *buf, size_t len, SSL *ssl, void *arg));
void **SSL_set_msg_callback_arg**(SSL *ctx, void *arg);
void **SSL_set_options**(SSL *ssl, unsigned long op);
void **SSL_set_quiet_shutdown**(SSL *ssl, int mode);
void **SSL_set_read_ahead**(SSL *ssl, int yes);
int **SSL_set_rfd**(SSL *ssl, int fd);
int **SSL_set_session**(SSL *ssl, SSL_SESSION *session);
void **SSL_set_shutdown**(SSL *ssl, int mode);
int **SSL_set_ssl_method**(SSL *ssl, const SSL_METHOD *meth);
void **SSL_set_time**(SSL *ssl, long t);
void **SSL_set_timeout**(SSL *ssl, long t);
void **SSL_set_verify**(SSL *ssl, int mode, int (*callback);(void))
void **SSL_set_verify_result**(SSL *ssl, long arg);
int **SSL_set_wfd**(SSL *ssl, int fd);
int **SSL_shutdown**(SSL *ssl);
int **SSL_state**(const SSL *ssl);
char ***SSL_state_string**(const SSL *ssl);
char ***SSL_state_string_long**(const SSL *ssl);
long **SSL_total_renegotiations**(SSL *ssl);
int **SSL_use_PrivateKey**(SSL *ssl, EVP_PKEY *pkey);
int **SSL_use_PrivateKey_ASN1**(int type, SSL *ssl, unsigned char *d, long len);
int **SSL_use_PrivateKey_file**(SSL *ssl, char *file, int type);
int **SSL_use_RSAPrivateKey**(SSL *ssl, RSA *rsa);
int **SSL_use_RSAPrivateKey_ASN1**(SSL *ssl, unsigned char *d, long len);
int **SSL_use_RSAPrivateKey_file**(SSL *ssl, char *file, int type);
int **SSL_use_certificate**(SSL *ssl, X509 *x);
int **SSL_use_certificate_ASN1**(SSL *ssl, int len, unsigned char *d);
int **SSL_use_certificate_file**(SSL *ssl, char *file, int type);
int **SSL_version**(const SSL *ssl);
int **SSL_want**(const SSL *ssl);
int **SSL_want_nothing**(const SSL *ssl);
int **SSL_want_read**(const SSL *ssl);
int **SSL_want_write**(const SSL *ssl);
int **SSL_want_x509_lookup**(const SSL *ssl);
int **SSL_write**(SSL *ssl, const void *buf, int num);
void **SSL_set_psk_client_callback**(SSL *ssl, unsigned int (*callback)(SSL *ssl, const char *hint, char *identity, unsigned int max_identity_len, unsigned char *psk, unsigned int max_psk_len));
int **SSL_use_psk_identity_hint**(SSL *ssl, const char *hint);
void **SSL_set_psk_server_callback**(SSL *ssl, unsigned int (*callback)(SSL *ssl, const char *identity, unsigned char *psk, int max_psk_len));
const char ***SSL_get_psk_identity_hint**(SSL *ssl);
const char ***SSL_get_psk_identity**(SSL *ssl);

SEE ALSO

openssl (1), *crypto* (3), *SSL_accept* (3), *SSL_clear* (3), *SSL_connect* (3), *SSL_CIPHER_get_name* (3), *SSL_COMP_add_compression_method* (3), *SSL_CTX_add_extra_chain_cert* (3), *SSL_CTX_add_session* (3), *SSL_CTX_ctrl* (3), *SSL_CTX_flush_sessions* (3), *SSL_CTX_get_ex_new_index* (3), *SSL_CTX_get_verify_mode* (3), *SSL_CTX_load_verify_locations* (3) *SSL_CTX_new* (3), *SSL_CTX_sess_number* (3), *SSL_CTX_sess_set_cache_size* (3), *SSL_CTX_sess_set_get_cb* (3), *SSL_CTX_sessions* (3), *SSL_CTX_set_cert_store* (3), *SSL_CTX_set_cert_verify_callback* (3), *SSL_CTX_set_cipher_list* (3), *SSL_CTX_set_client_CA_list* (3), *SSL_CTX_set_client_cert_cb* (3), *SSL_CTX_set_default_passwd_cb* (3), *SSL_CTX_set_generate_session_id* (3), *SSL_CTX_set_info_callback* (3), *SSL_CTX_set_max_cert_list* (3), *SSL_CTX_set_mode* (3), *SSL_CTX_set_msg_callback* (3), *SSL_CTX_set_options* (3), *SSL_CTX_set_quiet_shutdown* (3), *SSL_CTX_set_session_cache_mode* (3),

SSL_CTX_set_session_id_context (3), *SSL_CTX_set_ssl_version* (3), *SSL_CTX_set_timeout* (3), *SSL_CTX_set_tmp_rsa_callback* (3), *SSL_CTX_set_tmp_dh_callback* (3), *SSL_CTX_set_verify* (3), *SSL_CTX_use_certificate* (3), *SSL_alert_type_string* (3), *SSL_do_handshake* (3), *SSL_get_SSL_CTX* (3), *SSL_get_ciphers* (3), *SSL_get_client_CA_list* (3), *SSL_get_default_timeout* (3), *SSL_get_error* (3), *SSL_get_ex_data_X509_STORE_CTX_idx* (3), *SSL_get_ex_new_index* (3), *SSL_get_fd* (3), *SSL_get_peer_cert_chain* (3), *SSL_get_rbio* (3), *SSL_get_session* (3), *SSL_get_verify_result* (3), *SSL_get_version* (3), *SSL_library_init* (3), *SSL_load_client_CA_file* (3), *SSL_new* (3), *SSL_pending* (3), *SSL_read* (3), *SSL_rstate_string* (3), *SSL_session_reused* (3), *SSL_set_bio* (3), *SSL_set_connect_state* (3), *SSL_set_fd* (3), *SSL_set_session* (3), *SSL_set_shutdown* (3), *SSL_shutdown* (3), *SSL_state_string* (3), *SSL_want* (3), *SSL_write* (3), *SSL_SESSION_free* (3), *SSL_SESSION_get_ex_new_index* (3), *SSL_SESSION_get_time* (3), *d2i_SSL_SESSION* (3), *SSL_CTX_set_psk_client_callback* (3), *SSL_CTX_use_psk_identity_hint* (3), *SSL_get_psk_identity* (3)

HISTORY

The *ssl* (3) document appeared in OpenSSL 0.9.2

NAME

ssp — bounds checked libc functions

LIBRARY

library “libssp”

SYNOPSIS

```
#include <ssp/stdio.h>

int
sprintf(char *str, const char *fmt, ...);

int
vsprintf(char *str, const char *fmt, va_list ap);

int
snprintf(char *str, size_t len, const char *fmt, ...);

int
vsnprintf(char *str, size_t len, const char *fmt, va_list ap);

char *
gets(char *str);

char *
fgets(char *str, int len, FILE *fp);

#include <ssp/string.h>

void *
memcpy(void *str, const void *ptr, size_t len);

void *
memmove(void *str, const void *ptr, size_t len);

void *
memset(void *str, int val, size_t len);

char *
strcpy(char *str, const char *ptr, size_t len);

char *
strcat(char *str, const char *ptr, size_t len);

char *
strncpy(char *str, const char *ptr, size_t len);

char *
strncat(char *str, const char *ptr, size_t len);

#include <ssp/strings.h>

void *
bcopy(const void *ptr, void *str, size_t len);

void *
bzero(void *str, size_t len);

#include <ssp/unistd.h>
```

ssize_t
read(*int fd*, *void *str*, *size_t len*);

int
readlink(*const char * restrict path*, *char * restrict str*, *size_t len*);

int
getcwd(*char *str*, *size_t len*);

DESCRIPTION

When _FORTIFY_SOURCE bounds checking is enabled as described below, the above functions get overwritten to use the gcc(1) __builtin_object_size(3) function to compute the size of *str* if known at compile time and perform bounds check on it in order to avoid data buffer or stack buffer overflows. If an overflow is detected the routines will call abort(3).

To enable these function overrides the following should be added to the gcc(1) command line: "−I/usr/include/ssp" to override the standard include files and "−D_FORTIFY_SOURCE=1" or "−D_FORTIFY_SOURCE=2".

If _FORTIFY_SOURCE is set to 1 the code will compute the maximum possible buffer size for *str*, and if set to 2 it will compute the minimum buffer size.

SEE ALSO

gcc(1), read(2), readlink(2), __builtin_object_size(3), abort(3), getcwd(3), stdio(3), string(3)

HISTORY

The **ssp** library appeared NetBSD 4.0.

NAME

string_to_flags, **flags_to_string** — Stat flags parsing and printing functions

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

char *
flags_to_string(u_long flags, const char *def);

int
string_to_flags(char **stringp, u_long *setp, u_long clrp);
```

DESCRIPTION

The **flags_to_string**() and **string_to_flags**() functions are used by programs such as ls(1), mtree(8), makefs(8), etc., to parse and/or print the st_flags field in the stat(2) structure.

They recognize the following flags:

arch (SF_ARCHIVED)	file is archived.
nodump (UF_NODUMP)	do not dump file.
opaque (UF_OPAQUE)	directory is opaque in union filesystems.
sappnd (SF_APPEND)	writes to the file may only append (superuser only).
schg (SF_IMMUTABLE)	file cannot be changed, is immutable (superuser only).
snap (SF_SNAPSHOT)	file is snapshot inode.
uappnd (UF_APPEND)	writes to the file may only append (user only).
uchg (UF_IMMUTABLE)	file cannot be changed, is immutable (user only).

The **flags_to_string**() function converts the bits set in the *flags* argument to a comma-separated string and returns it. If no flags are set, then the *def* string is returned. The returned string is allocated via malloc(3) and it is the responsibility of the caller to free(3) it.

The **string_to_flags**() function takes a *stringp* of space, comma, or tab separated flag names and places their bit value on the *setp* argument. If the flag name is prefixed by: “no”, then the bit value is placed on the *clrp* argument.

RETURN VALUES

flags_to_string() returns the symbolic representation of flags, the default string, or NULL if allocation failed.

string_to_flags() returns 0 on success and 1 if it fails to parse the string, setting *stringp* to point to the first string that it failed to parse.

SEE ALSO

stat(2)

NAME

stdarg, **va_arg**, **va_copy**, **va_end**, **va_start** — variable argument lists

SYNOPSIS

```
#include <stdarg.h>

void
va_start(va_list ap, last);

type
va_arg(va_list ap, type);

void
va_copy(va_list dest, va_list src);

void
va_end(va_list ap);
```

DESCRIPTION

A function may be called with a varying number of arguments of varying types. The include file ⟨stdarg.h⟩ declares a type (*va_list*) and defines three macros for stepping through a list of arguments whose number and types are not known to the called function.

The called function must declare an object of type *va_list* which is used by the macros **va_start**(), **va_arg**(), **va_end**(), and, optionally, **va_copy**().

The **va_start**() macro initializes *ap* for subsequent use by **va_arg**(), **va_copy**() and **va_end**(), and must be called first.

The parameter *last* is the name of the last parameter before the variable argument list, i.e. the last parameter of which the calling function knows the type.

Because the address of this parameter is used in the **va_start**() macro, it should not be declared as a register variable, or as a function or an array type.

The **va_start**() macro returns no value.

The **va_arg**() macro expands to an expression that has the type and value of the next argument in the call. The parameter *ap* is the *va_list ap* initialized by **va_start**(). Each call to **va_arg**() modifies *ap* so that the next call returns the next argument. The parameter *type* is a type name specified so that the type of a pointer to an object that has the specified type can be obtained simply by adding a * to *type*.

If there is no next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), random errors will occur.

If the type in question is one that gets promoted, the promoted type should be used as the argument to **va_arg**(). The following describes which types are promoted (and to what):

- *short* is promoted to *int*
- *float* is promoted to *double*
- *char* is promoted to *int*

The first use of the **va_arg**() macro after that of the **va_start**() macro returns the argument after *last*. Successive invocations return the values of the remaining arguments.

The **va_copy**() macro makes *dest* a copy of *src* as if the **va_start**() macro had been applied to it followed by the same sequence of uses of the **va_arg**() macro as had previously been used to reach the present state of *src*.

The **va_copy**() macro returns no value.

The **va_end**() macro handles a normal return from the function whose variable argument list was initialized by **va_start**() or **va_copy**().

The **va_end**() macro returns no value.

EXAMPLES

The function **foo**() takes a string of format characters and prints out the argument associated with each format character based on the type.

```
void
foo(char *fmt, ...)
{
        va_list ap;
        int d, c;
        char *s;
        double f;

        va_start(ap, fmt);
        while (*fmt)
                switch (*fmt++) {
                case 's':                       /* string */
                        s = va_arg(ap, char *);
                        printf("string %s\n", s);
                        break;
                case 'd':                       /* int */
                        d = va_arg(ap, int);
                        printf("int %d\n", d);
                        break;
                case 'c':                       /* char */
                        c = va_arg(ap, int);    /* promoted */
                        printf("char %c\n", c);
                        break;
                case 'f':                       /* float */
                        f = va_arg(ap, double); /* promoted */
                        printf("float %f\n", f);
                }
        va_end(ap);
}
```

STANDARDS

The **va_start**(), **va_arg**(), **va_copy**(), and **va_end**() macros conform to ISO/IEC 9899:1999 (“ISO C99”).

HISTORY

The **va_start**(), **va_arg**() and **va_end**() macros were introduced in ANSI X3.159-1989 (“ANSI C89”). The **va_copy**() macro was introduced in ISO/IEC 9899:1999 (“ISO C99”).

COMPATIBILITY

These macros are *not* compatible with the historic macros they replace. A backward compatible version can be found in the include file ⟨varargs.h⟩.

BUGS

Unlike the *varargs* macros, the **stdarg** macros do not permit programmers to code a function with no fixed arguments. This problem generates work mainly when converting *varargs* code to **stdarg** code, but it also creates difficulties for variadic functions that wish to pass all of their arguments on to a function that takes a *va_list* argument, such as `vfprintf`(3).

NAME

stdio — standard input/output library functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

DESCRIPTION

The standard I/O library provides a simple and efficient buffered stream I/O interface. Input and output is mapped into logical data streams and the physical I/O characteristics are concealed. The functions and macros are listed below; more information is available from the individual man pages.

A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve creating a new file. Creating an existing file causes its former contents to be discarded. If a file can support positioning requests (such as a disk file, as opposed to a terminal) then a *file position indicator* associated with the stream is positioned at the start of the file (byte zero), unless the file is opened with append mode. If append mode is used, the position indicator will be placed the end-of-file. The position indicator is maintained by subsequent reads, writes and positioning requests. All input occurs as if the characters were read by successive calls to the `fgetc`(3) function; all output takes place as if all characters were read by successive calls to the `fputc`(3) function.

A file is disassociated from a stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transferred to the host environment) before the stream is disassociated from the file. The value of a pointer to a `FILE` object is indeterminate after a file is closed (garbage).

A file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at the start). If the main function returns to its original caller, or the `exit`(3) function is called, all open files are closed (hence all output streams are flushed) before program termination. Other methods of program termination, such as `abort`(3) do not bother about closing files properly.

This implementation needs and makes no distinction between “text” and “binary” streams. In effect, all streams are binary. No translation is performed and no extra padding appears on any stream.

At program startup, three streams are predefined and need not be opened explicitly:

- *standard input* (for reading conventional input),
- *standard output* (for writing conventional output), and
- *standard error* (for writing diagnostic output).

These streams are abbreviated *stdin*, *stdout* and *stderr*. Initially, the standard error stream is unbuffered; the standard input and output streams are fully buffered if and only if the streams do not refer to an interactive or “terminal” device, as determined by the `isatty`(3) function. In fact, *all* freshly-opened streams that refer to terminal devices default to line buffering, and pending output to such streams is written automatically whenever an such an input stream is read. Note that this applies only to “true reads”; if the read request can be satisfied by existing buffered data, no automatic flush will occur. In these cases, or when a large amount of computation is done after printing part of a line on an output terminal, it is necessary to `fflush`(3) the standard output before going off and computing so that the output will appear. Alternatively, these defaults may be modified via the `setvbuf`(3) function.

The **stdio** library is a part of the library libc.a and routines are automatically loaded as needed by compilers such as cc(1). The SYNOPSIS sections of the following manual pages indicate which include files are to be used, what the compiler declaration for the function looks like and which external variables are of interest.

In multi-threaded applications, operations on streams perform implicit locking, except for the *getc_unlocked*, *getchar_unlocked*, *putc_unlocked*, and *putchar_unlocked* functions. Explicit control of stream locking is available through the *flockfile*, *ftrylockfile*, and *funlockfile* functions .

The following are defined as macros; these names may not be re-used without first removing their current definitions with #undef: BUFSIZ, EOF, FILENAME_MAX, FOPEN_MAX, L_cuserid, L_ctermid, L_tmpnam, NULL, SEEK_END, SEEK_SET, SEE_CUR, TMP_MAX, **clearerr**(), **feof**(), **ferror**(), **fileno**(), **freopen**(), **fwopen**(), **getc**(), **getc_unlocked**(), **getchar**(), **getchar_unlocked**(), **putc**(), **putc_unlocked**(), **putchar**(), **putchar_unlocked**(), stderr, stdin, stdout. Function versions of the macro functions **feof**(), **ferror**(), **clearerr**(), **fileno**(), **getc**(), **getc_unlocked**(), **getchar**(), **getchar_unlocked**(), **putc**(), **putc_unlocked**(), **putchar**(), and **putchar_unlocked**() exist and will be used if the macros definitions are explicitly removed.

SEE ALSO

close(2), open(2), read(2), write(2)

STANDARDS

The **stdio** library conforms to ANSI X3.159-1989 (“ANSI C89”).

LIST OF FUNCTIONS

Function	Description
clearerr	check and reset stream status
fclose	close a stream
fdopen	stream open functions
feof	check and reset stream status
ferror	check and reset stream status
fflush	flush a stream
fgetc	get next character or word from input stream
fgetln	get a line from a stream
fgetpos	reposition a stream
fgets	get a line from a stream
fgetwc	get next wide character from input stream
fileno	check and reset stream status
flockfile	lock a stream
fopen	stream open functions
fprintf	formatted output conversion
fpurge	flush a stream
fputc	output a character or word to a stream
fputs	output a line to a stream
fputwc	output a wide character to a stream
fread	binary stream input/output
freopen	stream open functions
fropen	open a stream
fscanf	input format conversion
fseek	reposition a stream
fsetpos	reposition a stream
ftell	reposition a stream

ftrylockfile	lock a stream (non-blocking)
funlockfile	unlock a stream
funopen	open a stream
fwide	set/get orientation of a stream
fwopen	open a stream
fwrite	binary stream input/output
getc	get next character or word from input stream
getc_unlocked	get next character or word from input stream (no implicit locking)
getchar	get next character or word from input stream
getchar_unlocked	get next character or word from input stream (no implicit locking)
gets	get a line from a stream
getw	get next character or word from input stream
getwc	get next wide character from input stream
getwchar	get next wide character from input stream
mkstemp	create unique temporary file
mktemp	create unique temporary file
perror	system error messages
printf	formatted output conversion
putc	output a character or word to a stream
putc_unlocked	output a character or word to a stream (no implicit locking)
putchar	output a character or word to a stream
putchar_unlocked	output a character or word to a stream (no implicit locking)
puts	output a line to a stream
putw	output a character or word to a stream
putwc	output a wide character to a stream
putwchar	output a wide character to a stream
remove	remove directory entry
rewind	reposition a stream
scanf	input format conversion
setbuf	stream buffering operations
setbuffer	stream buffering operations
setlinebuf	stream buffering operations
setvbuf	stream buffering operations
snprintf	formatted output conversion
sprintf	formatted output conversion
sscanf	input format conversion
strerror	system error messages
sys_errlist	system error messages
sys_nerr	system error messages
tempnam	temporary file routines
tmpfile	temporary file routines
tmpnam	temporary file routines
ungetc	un-get character from input stream
ungetwc	un-get wide character from input stream
vfprintf	formatted output conversion
vfscanf	input format conversion
vprintf	formatted output conversion
vscanf	input format conversion

vsnprintf	formatted output conversion
vsprintf	formatted output conversion
vsscanf	input format conversion

BUGS

The standard buffered functions do not interact well with certain other library and system functions, especially `vfork`(2) and `abort`(3).

NAME

strcasecmp, **strncasecmp** — compare strings, ignoring case

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <strings.h>

int
strcasecmp(const char *s1, const char *s2);

int
strncasecmp(const char *s1, const char *s2, size_t len);
```

DESCRIPTION

The **strcasecmp**() and **strncasecmp**() functions compare the nul-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* after translation of each corresponding character to lower-case. The strings themselves are not modified. The comparison is done using unsigned characters, so that '\200' is greater than '\0'.

The **strncasecmp**() compares at most *len* characters.

SEE ALSO

bcmp(3), memcmp(3), strcmp(3), strcoll(3), strxfrm(3)

HISTORY

The **strcasecmp**() and **strncasecmp**() functions first appeared in 4.4BSD.

NOTES

If *len* is zero **strncasecmp**() returns always 0.

NAME

strcat, **strncat** — concatenate strings

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

char *
strcat(char * restrict s, const char * restrict append);

char *
strncat(char * restrict s, const char * restrict append, size_t count);
```

DESCRIPTION

The **strcat**() and **strncat**() functions append a copy of the nul-terminated string *append* to the end of the nul-terminated string *s*, then add a terminating '\0'. The string *s* must have sufficient space to hold the result.

The **strncat**() function appends not more than *count* characters where space for the terminating '\0' should not be included in *count*.

RETURN VALUES

The **strcat**() and **strncat**() functions return the pointer *s*.

EXAMPLES

The following appends "abc" to "chararray":

```
char *letters = "abcdefghi";

(void)strncat(chararray, letters, 3);
```

The following example shows how to use **strncat**() safely in conjunction with strncpy(3).

```
char buf[BUFSIZ];
char *input, *suffix;

(void)strncpy(buf, input, sizeof(buf) - 1);
buf[sizeof(buf) - 1] = '\0';
(void)strncat(buf, suffix, sizeof(buf) - 1 - strlen(buf));
```

The above will copy as many characters from "input" to "buf" as will fit. It then appends as many characters from suffix as will fit (or none if there is no space). For operations like this, the strlcpy(3) and strlcat(3) functions are a better choice, as shown below.

```
(void)strlcpy(buf, input, sizeof(buf));
(void)strlcat(buf, suffix, sizeof(buf));
```

SEE ALSO

bcopy(3), memccpy(3), memcpy(3), memmove(3), strcpy(3), strlcat(3), strlcpy(3)

STANDARDS

The **strcat**() and **strncat**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

NAME

strchr — locate character in string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

char *
strchr(const char *s, int c);
```

DESCRIPTION

The **strchr**() function locates the first occurrence of *c* in the string pointed to by *s*. The terminating NUL character is considered part of the string. If *c* is '\0', **strchr**() locates the terminating '\0'.

RETURN VALUES

The function **strchr**() returns a pointer to the located character, or NULL if the character does not appear in the string.

EXAMPLES

After the following call to **strchr**(), *p* will point to the string "oobar":

```
char *p;
char *s = "foobar";

p = strchr(s, 'o');
```

SEE ALSO

index(3), memchr(3), rindex(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **strchr**() function conforms to ANSI X3.159-1989 ("ANSI C89").

NAME

strcmp, **strncmp** — compare strings

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

int
strcmp(const char *s1, const char *s2);

int
strncmp(const char *s1, const char *s2, size_t len);
```

DESCRIPTION

The **strcmp**() and **strncmp**() functions lexicographically compare the nul-terminated strings *s1* and *s2*.

RETURN VALUES

The **strcmp**() and **strncmp**() return an integer greater than, equal to, or less than 0, according to whether the string *s1* is greater than, equal to, or less than the string *s2*. The comparison is done using unsigned characters, so that `\200` is greater than '`\0`'.

The **strncmp**() compares not more than *len* characters.

SEE ALSO

bcmp(3), memcmp(3), strcasecmp(3), strcoll(3), strxfrm(3)

STANDARDS

The **strcmp**() and **strncmp**() functions conform to ANSI X3.159-1989 ("ANSI C89").

NOTES

If *len* is zero **strncmp**() returns always 0.

NAME

strcoll — compare strings according to current collation

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

int
strcoll(const char *s1, const char *s2);
```

DESCRIPTION

The **strcoll**() function lexicographically compares the nul-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2*.

SEE ALSO

bcmp(3), memcmp(3), setlocale(3), strcasecmp(3), strcmp(3), strxfrm(3)

STANDARDS

The **strcoll**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

strcpy, **strncpy** — copy strings

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

char *
strcpy(char * restrict dst, const char * restrict src);

char *
strncpy(char * restrict dst, const char * restrict src, size_t len);
```

DESCRIPTION

The **strcpy**() and **strncpy**() functions copy the string *src* to *dst* (including the terminating '\0' character).

The **strncpy**() function copies not more than *len* characters into *dst*, appending '\0' characters if *src* is less than *len* characters long, and *not* terminating *dst* if *src* is *len* or more characters long.

RETURN VALUES

The **strcpy**() and **strncpy**() functions return *dst*.

EXAMPLES

The following sets "chararray" to "abc\0\0\0".

```
(void)strncpy(chararray, "abc", 6);
```

The following sets "chararray" to "abcdef" and does *not* nul-terminate *chararray* because the source string is >= the length parameter. **strncpy**() *only* nul-terminates the destination string when the length of the source string is less than the length parameter.

```
(void)strncpy(chararray, "abcdefgh", 6);
```

The following copies as many characters from *input* to *buf* as will fit and nul-terminates the result. Because **strncpy**() does *not* guarantee to nul-terminate the string itself, we must do this by hand.

```
char buf[BUFSIZ];

(void)strncpy(buf, input, sizeof(buf) - 1);
buf[sizeof(buf) - 1] = '\0';
```

Note that strlcpy(3) is a better choice for this kind of operation. The equivalent using strlcpy(3) is simply:

```
(void)strlcpy(buf, input, sizeof(buf));
```

SEE ALSO

bcopy(3), memccpy(3), memcpy(3), memmove(3), strlcpy(3)

STANDARDS

The **strcpy**() and **strncpy**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

NAME

strcspn — span the complement of a string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

size_t
strcspn(const char *s, const char *charset);
```

DESCRIPTION

The **strcspn**() function spans the initial part of the nul-terminated string *s* as long as the characters from *s* do not occur in string *charset* (it spans the *complement* of *charset*).

RETURN VALUES

The **strcspn**() function returns the number of characters spanned.

EXAMPLES

The following call to **strcspn**() will return 3, since the first three characters of string *s* do not occur in string *charset*:

```
char *s = "foobar";
char *charset = "bar";
size_t span;

span = strcspn(s, charset);
```

SEE ALSO

index(3), memchr(3), rindex(3), strchr(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **strcspn**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

strdup, **strndup** — save a copy of a string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

char *
strdup(const char *str);

char *
strndup(const char *str, size_t len);
```

DESCRIPTION

The **strdup**() function allocates sufficient memory for a copy of the string *str*, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3).

If insufficient memory is available, NULL is returned.

The **strndup**() function copies at most *len* characters from the string *str* always NUL terminating the copied string.

EXAMPLES

The following will point *p* to an allocated area of memory containing the nul-terminated string "foobar":

```
char *p;

if ((p = strdup("foobar")) == NULL) {
        fprintf(stderr, "Out of memory.\n");
        exit(1);
}
```

ERRORS

The **strdup**() function may fail and set the external variable *errno* for any of the errors specified for the library function malloc(3).

SEE ALSO

free(3), malloc(3), strcpy(3), strlen(3)

HISTORY

The **strdup**() function first appeared in 4.4BSD.

NAME

perror, **strerror**, **strerror_r**, **sys_errlist**, **sys_nerr** — system error messages

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

void
perror(const char *string);

#include <errno.h>

extern const char * const sys_errlist[];
extern const int sys_nerr;

#include <string.h>

char *
strerror(int errnum);

int
strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

DESCRIPTION

The **strerror**(), **strerror_r**(), and **perror**() functions look up the language-dependent error message string corresponding to an error number.

The **strerror**() function accepts an error number argument *errnum* and returns a pointer to the corresponding message string.

The **strerror_r**() function renders the same result into *strerrbuf* for a maximum of *buflen* characters and returns 0 upon success.

The **perror**() function finds the error message corresponding to the current value of the global variable *errno* (intro(2)) and writes it, followed by a newline, to the standard error file descriptor. If the argument *string* is non-NULL and does not point to the nul character, this string is prepended to the message string and separated from it by a colon and space (": "); otherwise, only the error message string is printed.

If the error number is not recognized, these functions pass an error message string containing "Unknown error: " followed by the error number in decimal. To warn about this, **strerror**() sets errno to EINVAL, and **strerror_r**() returns EINVAL. Error numbers recognized by this implementation fall in the range 0 < *errnum* < *sys_nerr*.

If insufficient storage is provided in *strerrbuf* (as specified in *buflen*) to contain the error string, **strerror_r**() returns ERANGE and *strerrbuf* will contain an error message that has been truncated and NUL terminated to fit the length specified by *buflen*.

The message strings can be accessed directly using the external array *sys_errlist*. The external value *sys_nerr* contains a count of the messages in *sys_errlist*. The use of these variables is deprecated; **strerror**() or **strerror_r**() should be used instead.

SEE ALSO

intro(2), psignal(3)

STANDARDS

The **perror**() and **strerror**() functions conform to ISO/IEC 9899:1999 (“ISO C99”). The **strerror_r**() function conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

HISTORY

The **strerror**() and **perror**() functions first appeared in 4.4BSD. The **strerror_r**() function first appeared in NetBSD 4.0.

BUGS

For unknown error numbers, the **strerror**() function will return its result in a static buffer which may be overwritten by subsequent calls.

The return type for **strerror**() is missing a type-qualifier; it should actually be *const char* *.

Programs that use the deprecated *sys_errlist* variable often fail to compile because they declare it inconsistently.

NAME

strfmon — convert monetary value to string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <monetary.h>

ssize_t
strfmon(char * restrict s, size_t maxsize, const char * restrict format,
    ...);
```

DESCRIPTION

The **strfmon**() function places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the **%** character. After the **%**, the following appear in sequence:

- Zero or more of the following flags:

  =*f* A '**=**' character followed by another character *f* which is used as the numeric fill character.

  ^ Do not use grouping characters, regardless of the current locale default.

  + Represent positive values by prefixing them with a positive sign, and negative values by prefixing them with a negative sign. This is the default.

  (Enclose negative values in parentheses.

  ! Do not include a currency symbol in the output.

  - Left justify the result. Only valid when a field width is specified.

- An optional minimum field width as a decimal number. By default, there is no minimum width.
- A '**#**' sign followed by a decimal number specifying the maximum expected number of digits after the radix character.
- A '**.**' character followed by a decimal number specifying the number of digits after the radix character.
- One of the following conversion specifiers:

  i The *double* argument is formatted as an international monetary amount.

  n The *double* argument is formatted as a national monetary amount.

  % A '%' character is written.

RETURN VALUES

If the total number of resulting bytes including the terminating NULL byte is not more than *maxsize*, **strfmon**() returns the number of bytes placed into the array pointed to by *s*, not including the terminating NULL byte. Otherwise, −1 is returned, the contents of the array are indeterminate, and *errno* is set to indicate the error.

ERRORS

The **strfmon**() function will fail if:

[E2BIG] Conversion stopped due to lack of space in the buffer.

[EINVAL] The format string is invalid.

[ENOMEM] Not enough memory for temporary buffers.

SEE ALSO

localeconv(3)

STANDARDS

The **strfmon**() function conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

AUTHORS

The **strfmon**() function was implemented by Alexey Zelkin ⟨phantom@FreeBSD.org⟩.

This manual page was written by Jeroen Ruigrok van der Werven ⟨asmodai@FreeBSD.org⟩ based on the standard’s text.

BUGS

The **strfmon**() function does not correctly handle multibyte characters in the *format* argument.

NAME

strftime — format date and time

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <time.h>

size_t
strftime(char * restrict buf, size_t maxsize, const char * restrict format,
    const struct tm * restrict timeptr);
```

DESCRIPTION

The **strftime**() function formats the information from *timeptr* into the buffer *buf* according to the string pointed to by *format*.

The *format* string consists of zero or more conversion specifications and ordinary characters. All ordinary characters are copied directly into the buffer. A conversion specification consists of a percent sign '%' and one other character.

No more than *maxsize* characters will be placed into the array. If the total number of resulting characters, including the terminating null character, is not more than *maxsize*, **strftime**() returns the number of characters in the array, not counting the terminating null. Otherwise, zero is returned and the contents of the array are undefined.

Each conversion specification is replaced by the characters as follows which are then copied into the buffer.

%A is replaced by the locale's full weekday name.

%a is replaced by the locale's abbreviated weekday name.

%B is replaced by the locale's full month name.

%b or **%h**
is replaced by the locale's abbreviated month name.

%C is replaced by the century (a year divided by 100 and truncated to an integer) as a decimal number [00,99].

%c is replaced by the locale's appropriate date and time representation.

%D is replaced by the date in the format "%m/%d/%y".

%d is replaced by the day of the month as a decimal number [01,31].

%e is replaced by the day of month as a decimal number [1,31]; single digits are preceded by a blank.

%F is replaced by the date in the format "%Y-%m-%d" (the ISO 8601 date format).

%G is replaced by the ISO 8601 year with century as a decimal number.

%g is replaced by the ISO 8601 year without century as a decimal number (00-99). This is the year that includes the greater part of the week. (Monday as the first day of a week). See also the '%V' conversion specification.

%H is replaced by the hour (24-hour clock) as a decimal number [00,23].

%I is replaced by the hour (12-hour clock) as a decimal number [01,12].

%j is replaced by the day of the year as a decimal number [001,366].

%k is replaced by the hour (24-hour clock) as a decimal number [0,23]; single digits are preceded by a blank.

%l is replaced by the hour (12-hour clock) as a decimal number [1,12]; single digits are preceded by a blank.

%M is replaced by the minute as a decimal number [00,59].

%m is replaced by the month as a decimal number [01,12].

%n is replaced by a newline.

%p is replaced by the locale's equivalent of either "AM" or "PM".

%R is replaced by the time in the format "`%H:%M`".

%r is replaced by the locale's representation of 12-hour clock time using AM/PM notation.

%S is replaced by the second as a decimal number [00,61]. The range of seconds is (00-61) instead of (00-59) to allow for the periodic occurrence of leap seconds and double leap seconds.

%s is replaced by the number of seconds since the Epoch, UTC (see `mktime`(3)).

%T is replaced by the time in the format "`%H:%M:%S`".

%t is replaced by a tab.

%U is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number [00,53].

%u is replaced by the weekday (Monday as the first day of the week) as a decimal number [1,7].

%V is replaced by the week number of the year (Monday as the first day of the week) as a decimal number [01,53]. According to ISO 8601 the week containing January 1 is week 1 if it has four or more days in the new year, otherwise it is week 53 of the previous year, and the next week is week 1. The year is given by the '`%G`' conversion specification.

%v is replaced by the date in the format "`%e-%b-%Y`".

%W is replaced by the week number of the year (Monday as the first day of the week) as a decimal number [00,53].

%w is replaced by the weekday (Sunday as the first day of the week) as a decimal number [0,6].

%X is replaced by the locale's appropriate time representation.

%x is replaced by the locale's appropriate date representation.

%Y is replaced by the year with century as a decimal number.

%y is replaced by the year without century as a decimal number [00,99].

%Z is replaced by the time zone name.

%z is replaced by the offset from ITC in the ISO 8601 format "`[-]hhmm`".

%% is replaced by '`%`'.

SEE ALSO

`date`(1), `printf`(1), `ctime`(3), `printf`(3), `strptime`(3)

STANDARDS

The **strftime**() function conforms to ISO/IEC 9899:1999 ("ISO C99"). The '%C', '%D', '%e', '%g', '%G', '%h', '%k', '%l', '%n', '%r', '%R', '%s', '%t', '%T', '%u', '%V', and '%v' conversion specifications are extensions.

Use of the ISO 8601 conversions may produce non-intuitive results. Week 01 of a year is per definition the first week which has the Thursday in this year, which is equivalent to the week which contains the fourth day of January. In other words, the first week of a new year is the week which has the majority of its days in the new year. Week 01 might also contain days from the previous year and the week before week 01 of a year is the last week (52 or 53) of the previous year even if it contains days from the new year. A week starts with Monday (day 1) and ends with Sunday (day 7). For example, the first week of the year 1997 lasts from 1996-12-30 to 1997-01-05.

BUGS

There is no conversion specification for the phase of the moon.

NAME

strcat, **strlcat**, **strncat**, **strchr**, **strrchr**, **strcmp**, **strncmp**, **strcoll**, **strcpy**, **strlcpy**, **strncpy**, **strerror**, **strerror_r**, **strlen**, **strpbrk**, **strsep**, **stresep**, **strspn**, **strcspn**, **strdup**, **strndup**, **strstr**, **strcasestr**, **strtok**, **strtok_r**, **strxfrm** — string specific functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

char *
strcat(char *s, const char * append);

size_t
strlcat(char *dst, const char *src, size_t size);

char *
strncat(char *s, const char *append, size_t count);

char *
strchr(const char *s, int c);

char *
strrchr(const char *s, int c);

int
strcmp(const char *s1, const char *s2);

int
strncmp(const char *s1, const char *s2, size_t count);

int
strcoll(const char *s1, const char *s2);

char *
strcpy(char *dst, const char *src);

size_t
strlcpy(char *dst, const char *src, size_t size);

char *
strncpy(char *dst, const char *src, size_t count);

char *
strerror(int errno);

int
strerror_r(int errnum, char *strerrbuf, size_t buflen);

size_t
strlen(const char *s);

char *
strpbrk(const char *s, const char *charset);

char *
strsep(char **stringp, const char *delim);
```

char ∗
stresep(*char ∗∗stringp*, *const char ∗delim*, *int escape*);

size_t
strspn(*const char ∗s*, *const char ∗charset*);

size_t
strcspn(*const char ∗s*, *const char ∗charset*);

char ∗
strdup(*const char ∗str*);

char ∗
strndup(*const char ∗str*, *size_t len*);

char ∗
strstr(*const char ∗big*, *const char ∗little*);

char ∗
strcasestr(*const char ∗big*, *const char ∗little*);

char ∗
strtok(*char ∗s*, *const char ∗delim*);

char ∗
strtok_r(*char ∗s*, *const char ∗delim*, *char ∗∗lasts*);

size_t
strxfrm(*char ∗dst*, *const char ∗src*, *size_t n*);

DESCRIPTION

The string functions manipulate strings terminated by a nul byte.

See the specific manual pages for more information. For manipulating variable length generic objects as byte strings (without the nul byte check), see bstring(3).

Except as noted in their specific manual pages, the string functions do not test the destination for size limitations.

SEE ALSO

bstring(3), strcat(3), strchr(3), strcmp(3), strcoll(3), strcpy(3), strcspn(3), strdup(3), strerror(3), strings(3), strlcat(3), strlen(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3), strxfrm(3)

STANDARDS

The **strcat**(), **strncat**(), **strchr**(), **strrchr**(), **strcmp**(), **strncmp**(), **strcpy**(), **strncpy**(), **strcoll**(), **strerror**(), **strlen**(), **strpbrk**(), **strsep**(), **strspn**(), **strcspn**(), **strstr**(), **strtok**(), and **strxfrm**() functions conform to ANSI X3.159-1989 (“ANSI C89”).

The **strtok_r**() function conforms to IEEE Std 1003.1c-1995 (“POSIX.1”).

The **strerror_r**() function conform to IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

stringlist, **sl_init**, **sl_add**, **sl_free**, **sl_find**, **sl_delete** — stringlist manipulation functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stringlist.h>

StringList *
sl_init();

int
sl_add(StringList *sl, char *item);

void
sl_free(StringList *sl, int freeall);

char *
sl_find(StringList *sl, const char *item);

int
sl_delete(StringList *sl, const char *item, int freeit);
```

DESCRIPTION

The **stringlist** functions manipulate stringlists, which are lists of strings that extend automatically if necessary.

The *StringList* structure has the following definition:

```
typedef struct _stringlist {
        char   **sl_str;
        size_t   sl_max;
        size_t   sl_cur;
} StringList;
```

sl_str a pointer to the base of the array containing the list.

sl_max the size of *sl_str*.

sl_cur the offset in *sl_str* of the current element.

The following stringlist manipulation functions are available:

sl_init() Create a stringlist. Returns a pointer to a *StringList*, or NULL in case of failure.

sl_free() Releases memory occupied by *sl* and the *sl->sl_str* array. If *freeall* is non-zero, then each of the items within *sl->sl_str* is released as well.

sl_add() Add *item* to *sl->sl_str* at *sl->sl_cur*, extending the size of *sl->sl_str*. Returns zero upon success, −1 upon failure.

sl_find() Find *item* in *sl*, returning NULL if it's not found.

sl_delete()
Remove *item* from the list. If *freeit* is non-zero, the string is freed. Returns 0 if the name is found and −1 if the name is not found.

SEE ALSO

`free`(3), `malloc`(3)

HISTORY

The **`stringlist`** functions appeared in NetBSD 1.3.

NAME

bcmp, **bcopy**, **bzero**, **ffs**, **index**, **rindex**, **strcasecmp**, **strncasecmp** — string operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <strings.h>

int
bcmp(const void *b1, const void *b2, size_t len);

void
bcopy(const void *src, void *dst, size_t len);

void
bzero(void *b, size_t len);

int
ffs(int value);

char *
index(const char *s, int c);

char *
rindex(const char *s, int c);

int
strcasecmp(const char *s1, const char *s2);

int
strncasecmp(const char *s1, const char *s2, size_t len);
```

DESCRIPTION

These functions all live in the `strings.h` header file. Except for **ffs**(), they operate on strings. **index**(), **rindex**(), and **strcasecmp**() need nul-terminated strings.

See the specific manual pages for more information.

See `string`(3) for string functions that follow ANSI X3.159-1989 (“ANSI C89”) or ISO/IEC 9899:1999 (“ISO C99”), `bstring`(3) for functions that operate on strings that are not nul-terminated, and `bitstring`(3) for bit-string manipulation macros.

SEE ALSO

`bcmp`(3), `bcopy`(3), `bitstring`(3), `bstring`(3), `bzero`(3), `ffs`(3), `index`(3), `rindex`(3), `strcasecmp`(3), `string`(3)

NAME

strlcpy, **strlcat** — size-bounded string copying and concatenation

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

size_t
strlcpy(char *dst, const char *src, size_t size);

size_t
strlcat(char *dst, const char *src, size_t size);
```

DESCRIPTION

The **strlcpy**() and **strlcat**() functions copy and concatenate strings respectively. They are designed to be safer, more consistent, and less error prone replacements for strncpy(3) and strncat(3). Unlike those functions, **strlcpy**() and **strlcat**() take the full size of the buffer (not just the length) and guarantee to NUL-terminate the result (as long as *size* is larger than 0 or, in the case of **strlcat**(), as long as there is at least one byte free in *dst*). Note that you should include a byte for the NUL in *size*. Also note that **strlcpy**() and **strlcat**() only operate on true "C" strings. This means that for **strlcpy**() *src* must be NUL-terminated and for **strlcat**() both *src* and *dst* must be NUL-terminated.

The **strlcpy**() function copies up to *size* - 1 characters from the NUL-terminated string *src* to *dst*, NUL-terminating the result.

The **strlcat**() function appends the NUL-terminated string *src* to the end of *dst*. It will append at most *size* - strlen(dst) - 1 bytes, NUL-terminating the result.

RETURN VALUES

The **strlcpy**() and **strlcat**() functions return the total length of the string they tried to create. For **strlcpy**() that means the length of *src*. For **strlcat**() that means the initial length of *dst* plus the length of *src*. While this may seem somewhat confusing it was done to make truncation detection simple.

Note however, that if **strlcat**() traverses *size* characters without finding a NUL, the length of the string is considered to be *size* and the destination string will not be NUL-terminated (since there was no space for the NUL). This keeps **strlcat**() from running off the end of a string. In practice this should not happen (as it means that either *size* is incorrect or that *dst* is not a proper "C" string). The check exists to prevent potential security problems in incorrect code.

EXAMPLES

The following code fragment illustrates the simple case:

```
char *s, *p, buf[BUFSIZ];

...

(void)strlcpy(buf, s, sizeof(buf));
(void)strlcat(buf, p, sizeof(buf));
```

To detect truncation, perhaps while building a pathname, something like the following might be used:

```
char *dir, *file, pname[MAXPATHLEN];

...
```

```
if (strlcpy(pname, dir, sizeof(pname)) ≥ sizeof(pname))
        goto toolong;
if (strlcat(pname, file, sizeof(pname)) ≥ sizeof(pname))
        goto toolong;
```

Since we know how many characters we copied the first time, we can speed things up a bit by using a copy instead of an append:

```
char *dir, *file, pname[MAXPATHLEN];
size_t n;

...

n = strlcpy(pname, dir, sizeof(pname));
if (n ≥ sizeof(pname))
        goto toolong;
if (strlcpy(pname + n, file, sizeof(pname) - n) ≥ sizeof(pname) - n)
        goto toolong;
```

However, one may question the validity of such optimizations, as they defeat the whole purpose of **strlcpy**() and **strlcat**().

SEE ALSO

snprintf(3), strncat(3), strncpy(3)

HISTORY

strlcpy() and **strlcat**() first appeared in OpenBSD 2.4, then in NetBSD 1.4.3 and FreeBSD 3.3.

NAME

strlen — find length of string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

size_t
strlen(const char *s);
```

DESCRIPTION

The **strlen**() function computes the length of the string *s*.

RETURN VALUES

The **strlen**() function returns the number of characters that precede the terminating NUL character.

SEE ALSO

string(3)

STANDARDS

The **strlen**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

strmode — convert inode status information into a symbolic string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

void
strmode(mode_t mode, char *bp);
```

DESCRIPTION

The **strmode**() function converts a file *mode* (the type and permission information associated with an inode, see stat(2)) into a symbolic string which is stored in the location referenced by *bp*. This stored string is eleven characters in length plus a trailing nul byte.

The first character is the inode type, and will be one of the following:

−	regular file
a	regular file in archive state 1
A	regular file in archive state 2
b	block special
c	character special
d	directory
l	symbolic link
p	fifo
s	socket
w	whiteout
?	unknown inode type

The next nine characters encode three sets of permissions, in three characters each. The first three characters are the permissions for the owner of the file, the second three for the group the file belongs to, and the third for the ‘‘other’’, or default, set of users.

Permission checking is done as specifically as possible. If read permission is denied to the owner of a file in the first set of permissions, the owner of the file will not be able to read the file. This is true even if the owner is in the file’s group and the group permissions allow reading or the ‘‘other’’ permissions allow reading.

If the first character of the three character set is an ‘‘r’’, the file is readable for that set of users; if a dash ‘‘−’’, it is not readable.

If the second character of the three character set is a ‘‘w’’, the file is writable for that set of users; if a dash ‘‘−’’, it is not writable.

The third character is the first of the following characters that apply:

S If the character is part of the owner permissions and the file is not executable or the directory is not searchable by the owner, and the set-user-id bit is set.

S If the character is part of the group permissions and the file is not executable or the directory is not searchable by the group, and the set-group-id bit is set.

T If the character is part of the other permissions and the file is not executable or the directory is not searchable by others, and the ‘‘sticky’’ (S_ISVTX) bit is set.

s If the character is part of the owner permissions and the file is executable or the directory searchable by the owner, and the set-user-id bit is set.

s If the character is part of the group permissions and the file is executable or the directory searchable by the group, and the set-group-id bit is set.

t If the character is part of the other permissions and the file is executable or the directory searchable by others, and the ‘‘sticky’’ (`S_ISVTX`) bit is set.

x The file is executable or the directory is searchable.

− None of the above apply.

The last character is a plus sign ‘‘+’’ if there are any alternative or additional access control methods associated with the inode, otherwise it will be a space.

Archive state 1 and archive state 2 represent file system dependent archive state for a file. Most file systems do not retain file archive state, and so will not report files in either archive state. msdosfs will report a file in archive state 1 if it has been archived more recently than modified. Hierarchical storage systems may have multiple archive states for a file and may define archive states 1 and 2 as appropriate.

SEE ALSO

`chmod`(1), `find`(1), `stat`(2), `getmode`(3), `setmode`(3)

HISTORY

The **`strmode`**() function first appeared in 4.4BSD.

NAME

strpbrk — locate multiple characters in string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

char *
strpbrk(const char *s, const char *charset);
```

DESCRIPTION

The **strpbrk**() function locates in the nul-terminated string *s* the first occurrence of any character in the string *charset* and returns a pointer to this character. If no characters from *charset* occur anywhere in *s* **strpbrk**() returns NULL.

SEE ALSO

index(3), memchr(3), rindex(3), strchr(3), strcspn(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **strpbrk**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

strptime — converts a character string to a time value

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <time.h>

char *
strptime(const char * restrict buf, const char * restrict format,
    struct tm * restrict tm);
```

DESCRIPTION

The **strptime**() function converts the character string pointed to by *buf* to values which are stored in the *tm* structure pointed to by *tm*, using the format specified by *format*.

The *format* string consists of zero or more conversion specifications, whitespace characters as defined by **isspace**(), and ordinary characters. All ordinary characters in *format* are compared directly against the corresponding characters in *buf*; comparisons which fail will cause **strptime**() to fail. Whitespace characters in *format* match any number of whitespace characters in *buf*, including none.

A conversion specification consists of a percent sign '%' followed by one or two conversion characters which specify the replacement required. There must be white-space or other non-alphanumeric characters between any two conversion specifications.

Conversion of alphanumeric strings (such as month and weekday names) is done without regard to case. Conversion specifications which cannot be matched will cause **strptime**() to fail.

The LC_TIME category defines the locale values for the conversion specifications. The following conversion specifications are supported:

%a the day of week, using the locale's weekday names; either the abbreviated or full name may be specified.

%A the same as **%a**.

%b the month, using the locale's month names; either the abbreviated or full name may be specified.

%B the same as **%b**.

%c the date and time, using the locale's date and time format.

%C the century number [0,99]; leading zeros are permitted but not required. This conversion should be used in conjunction with the %y conversion.

%d the day of month [1,31]; leading zeros are permitted but not required.

%D the date as %m/%d/%y.

%e the same as **%d**.

%F the date as %Y-%m-%d (the ISO 8601 date format).

%h the same as **%b**.

%H the hour (24-hour clock) [0,23]; leading zeros are permitted but not required.

%I the hour (12-hour clock) [1,12]; leading zeros are permitted but not required.

%j the day number of the year [1,366]; leading zeros are permitted but not required.

%k the same as **%H**.

%l the same as **%I**.

%m the month number [1,12]; leading zeros are permitted but not required.

%M the minute [0,59]; leading zeros are permitted but not required.

%n any white-space, including none.

%p the locale's equivalent of a.m. or p.m.

%r the time (12-hour clock) with %p, using the locale's time format.

%R the time as %H:%M.

%S the seconds [0,61]; leading zeros are permitted but not required.

%t any white-space, including none.

%T the time as %H:%M:%S.

%U the week number of the year (Sunday as the first day of the week) as a decimal number [0,53]; leading zeros are permitted but not required. All days in a year preceding the first Sunday are considered to be in week 0.

%w the weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required.

%W the week number of the year (Monday as the first day of the week) as a decimal number [0,53]; leading zeros are permitted but not required. All days in a year preceding the first Monday are considered to be in week 0.

%x the date, using the locale's date format.

%X the time, using the locale's time format.

%y the year within the 20th century [69,99] or the 21st century [0,68]; leading zeros are permitted but not required. If specified in conjunction with %C, specifies the year [0,99] within that century.

%Y the year, including the century (i.e., 1996).

%Z timezone name or no characters when time zone information is unavailable. (A NetBSD extension.)

%% matches a literal '%'. No argument is converted.

Modified conversion specifications

For compatibility, certain conversion specifications can be modified by the **E** and **O** modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specification. As there are currently neither alternative formats nor specifications supported by the system, the behavior will be as if the unmodified conversion specification were used.

Case is ignored when matching string items in *buf*, such as month and weekday names.

RETURN VALUES

If successful, the **strptime**() function returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned.

SEE ALSO

`ctime`(3), `isspace`(3), `localtime`(3), `strftime`(3)

STANDARDS

The **`strptime`**() function conforms to X/Open Portability Guide Issue 4 (“XPG4”).

BUGS

The **`%Z`** format specifier only accepts timezone abbreviations of the local timezone, or the value “GMT”. This limitation is caused by the ambiguity of overloaded timezone abbreviations, for example EST is both Eastern Standard Time and Eastern Australia Summer Time.

NAME

strrchr — locate character in string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

char *
strrchr(const char *s, int c);
```

DESCRIPTION

The **strrchr**() function locates the last occurrence of *c* (converted to a char) in the string *s*. If *c* is '\0', **strrchr**() locates the terminating '\0'.

RETURN VALUES

The **strrchr**() function returns a pointer to the character, or a null pointer if *c* does not occur anywhere in *s*.

EXAMPLES

After the following call to **strrchr**(), *p* will point to the string "obar":

```
char *p;
char *s = "foobar";

p = strrchr(s, 'o');
```

SEE ALSO

index(3), memchr(3), rindex(3), strchr(3), strcspn(3), strpbrk(3), strsep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **strrchr**() function conforms to ANSI X3.159-1989 ("ANSI C89").

NAME

strsep, **stresep** — separate strings

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

char *
strsep(char **stringp, const char *delim);

char *
stresep(char **stringp, const char *delim, int escape);
```

DESCRIPTION

The **strsep**() function locates, in the nul-terminated string referenced by **stringp*, the first occurrence of any character in the string *delim* (or the terminating '\0' character) and replaces it with a '\0'. The location of the next character after the delimiter character (or NULL, if the end of the string was reached) is stored in **stringp*. The original value of **stringp* is returned.

An "empty" field, i.e., one caused by two adjacent delimiter characters, can be detected by comparing the location referenced by the pointer returned by **strsep**() to '\0'.

If **stringp* is initially NULL, **strsep**() returns NULL. The **stresep**() function also takes an escape character that allows quoting the delimiter character so that it can be part of the source string.

EXAMPLES

The following uses **strsep**() to parse a string, containing tokens delimited by white space, into an argument vector:

```
char **ap, *argv[10], *inputstring;

for (ap = argv; ap < &argv[9] &&
    (*ap = strsep(&inputstring, " \t")) != NULL;) {
       if (**ap != '\0')
              ap++;
}
```

HISTORY

The **strsep**() function is intended as a replacement for the **strtok**() function. While the **strtok**() function should be preferred for portability reasons (it conforms to ANSI X3.159-1989 ("ANSI C89")) it is unable to handle empty fields, i.e., detect fields delimited by two adjacent delimiter characters, or to be used for more than a single string at a time. The **strsep**() function first appeared in 4.4BSD.

NAME

strsignal — get signal description string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

char *
strsignal(int sig);
```

DESCRIPTION

The **strsignal**() function returns a pointer to the language-dependent string describing a signal.

The array pointed to is not to be modified by the program, but may be overwritten by subsequent calls to **strsignal**().

SEE ALSO

intro(2), psignal(3), setlocale(3)

NAME

strspn — span a string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

size_t
strspn(const char *s, const char *charset);
```

DESCRIPTION

The **strspn**() function spans the initial part of the nul-terminated string *s* as long as the characters from *s* occur in string *charset*.

RETURN VALUES

The **strspn**() function returns the number of characters spanned.

EXAMPLES

The following call to **strspn**() will return 3, since the first three characters of string *s* are part of string *charset*:

```
char *s = "foobar";
char *charset = "of";
size_t span;

span = strspn(s, charset);
```

SEE ALSO

index(3), memchr(3), rindex(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strstr(3), strtok(3)

STANDARDS

The **strspn**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

strstr, **strcasestr** — locate a substring in a string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

char *
strstr(const char *big, const char *little);

char *
strcasestr(const char *big, const char *little);
```

DESCRIPTION

The **strstr**() function locates the first occurrence of the nul-terminated string *little* in the nul-terminated string *big*.

The **strcasestr**() function is similar to **strstr**(), but ignores the case of both strings.

RETURN VALUES

If *little* is an empty string, *big* is returned; if *little* occurs nowhere in *big*, NULL is returned; otherwise a pointer to the first character of the first occurrence of *little* is returned.

EXAMPLES

The following sets the pointer *ptr* to the "Bar Baz" portion of *largestring*:

```
const char *largestring = "Foo Bar Baz";
const char *smallstring = "Bar";
char *ptr;

ptr = strstr(largestring, smallstring);
```

SEE ALSO

index(3), memchr(3), rindex(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strtok(3)

STANDARDS

The **strstr**() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

NAME

strsuftoll, **strsuftollx** — convert a string to a long long, with suffix parsing

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

long long
strsuftoll(const char *desc, const char *val, long long min, long long max);

long long
strsuftollx(const char *desc, const char *val, long long min,
    long long max, char *errbuf, size_t errbuflen);
```

DESCRIPTION

The functions **strsuftoll**() and **strsuftollx**() convert *val* into a long long number, checking that the result is not smaller than *min* or larger than *max*. Two or more decimal numbers may be separated by an "x" to indicate a product. Each decimal number may have one of the following optional suffixes:

- b Block; multiply by 512
- k Kibi; multiply by 1024 (1 KiB)
- m Mebi; multiply by 1048576 (1 MiB)
- g Gibi; multiply by 1073741824 (1 GiB)
- t Tebi; multiply by 1099511627776 (1 TiB)
- w Word; multiply by the number of bytes in an integer

In the case of an error (range overflow or an invalid number), **strsuftollx**() places an error message into *errbuf* (which is *errbuflen* bytes long) and returns 0, and **strsuftoll**() displays that error and terminates the process.

RETURN VALUES

The functions **strsuftoll**() and **strsuftollx**() return either the result of the conversion, unless the value overflows or is not a number; in the latter case, **strsuftoll**() displays an error message and terminates the process with exit code 1, and **strsuftollx**() returns with 0 and *errbuf* contains a non-empty error message.

ERRORS

[ERANGE] The given string was out of range; the value converted has been clamped.

SEE ALSO

errx(3), strtoll(3)

BUGS

Ignores the current locale.

NAME

strtod, **strtof**, **strtold** — convert ASCII string to double, float, or long double

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

double
strtod(const char * restrict nptr, char ** restrict endptr);

float
strtof(const char * restrict nptr, char ** restrict endptr);

long double
strtold(const char * restrict nptr, char ** restrict endptr);
```

DESCRIPTION

The **strtod**() function converts the initial portion of the string pointed to by *nptr* to *double* representation.

The **strtof**() function converts the initial portion of the string pointed to by *nptr* to *float* representation.

The **strtold**() function converts the initial portion of the string pointed to by *nptr* to *long double* representation.

The expected form of the string is an optional plus (‘+’) or minus sign (‘−’) followed by one of the following:

- a sequence of digits optionally containing a decimal-point character, optionally followed by an exponent. An exponent consists of an ‘E’ or ‘e’, followed by an optional plus or minus sign, followed by a sequence of digits.
- one of INF or INFINITY, ignoring case.
- one of NAN or NAN(n-char-sequence-opt), ignoring case. This implementation currently does not interpret such a sequence.

Leading white-space characters in the string (as defined by the isspace(3) function) are skipped.

RETURN VALUES

The **strtod**(), **strtof**(), and **strtold**() functions return the converted value, if any.

A character sequence INF or INFINITY is converted to ∞, if supported, else to the largest finite floating-point number representable on the machine (i.e., VAX).

A character sequence NAN or NAN(n-char-sequence-opt) is converted to a quiet *NaN*, if supported, else remains unrecognized (i.e., VAX).

If *endptr* is not NULL, a pointer to the character after the last character used in the conversion is stored in the location referenced by *endptr*.

If no conversion is performed, zero is returned and the value of *nptr* is stored in the location referenced by *endptr*.

If the correct value would cause overflow, plus or minus HUGE_VAL, HUGE_VALF, or HUGE_VALL is returned (according to the return type and sign of the value), and ERANGE is stored in *errno*. If the correct value would cause underflow, zero is returned and ERANGE is stored in *errno*.

ERRORS

[ERANGE] Overflow or underflow occurred.

SEE ALSO

atof(3), atoi(3), atol(3), math(3), strtol(3), strtoul(3)

STANDARDS

The **strtod**() function conforms to ANSI X3.159-1989 (“ANSI C89”). The **strtof**() and **strtold**() functions conform to ISO/IEC 9899:1999 (“ISO C99”).

HISTORY

The **strtof**() and **strtold**() functions appeared in NetBSD 4.0.

NAME

strtok, strtok_r — string tokens

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

char *
strtok(char * restrict str, const char * restrict sep);

char *
strtok_r(char *str, const char *sep, char **lasts);
```

DESCRIPTION

The **strtok**() function is used to isolate sequential tokens in a nul-terminated string, *str*. These tokens are separated in the string by at least one of the characters in *sep*. The first time that **strtok**() is called, *str* should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead. The separator string, *sep*, must be supplied each time, and may change between calls.

The **strtok**() function returns a pointer to the beginning of each subsequent token in the string, after replacing the separator character itself with a NUL character. Separator characters at the beginning of the string or at the continuation point are skipped so that zero length tokens are not returned. When no more tokens remain, a null pointer is returned.

The **strtok_r**() function implements the functionality of **strtok**() but is passed an additional argument, *lasts*, which points to a user-provided pointer which is used by **strtok_r**() to store state which needs to be kept between calls to scan the same string; unlike **strtok**(), it is not necessary to delineate tokenizing to a single string at a time when using **strtok_r**().

EXAMPLES

The following will construct an array of pointers to each individual word in the string *s*:

```
#define MAXTOKENS     128

char s[512], *p, *tokens[MAXTOKENS];
char *last;
int i = 0;

snprintf(s, sizeof(s), "cat dog horse cow");

for ((p = strtok_r(s, " ", &last)); p;
    (p = strtok_r(NULL, " ", &last)), i++) {
        if (i < MAXTOKENS - 1)
                tokens[i] = p;
}
tokens[i] = NULL;
```

That is, `tokens[0]` will point to "cat", `tokens[1]` will point to "dog", `tokens[2]` will point to "horse", and `tokens[3]` will point to "cow".

SEE ALSO

index(3), memchr(3), rindex(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3)

STANDARDS

The **strtok**() function conforms to ANSI X3.159-1989 ("ANSI C89"). The **strtok_r**() function conforms to IEEE Std 1003.1c-1995 ("POSIX.1").

BUGS

The System V **strtok**(), if handed a string containing only delimiter characters, will not alter the next starting point, so that a call to **strtok**() with a different (or empty) delimiter string may return a non-NULL value. Since this implementation always alters the next starting point, such a sequence of calls would always return NULL.

NAME

strtol, **strtoll**, **strtoimax**, **strtoq** — convert string value to a long, long long, intmax_t or quad_t integer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>
#include <limits.h>

long int
strtol(const char * restrict nptr, char ** restrict endptr, int base);

long long int
strtoll(const char * restrict nptr, char ** restrict endptr, int base);

#include <inttypes.h>

intmax_t
strtoimax(const char * restrict nptr, char ** restrict endptr, int base);

#include <sys/types.h>
#include <stdlib.h>
#include <limits.h>

quad_t
strtoq(const char * restrict nptr, char ** restrict endptr, int base);
```

DESCRIPTION

The **strtol**() function converts the string in *nptr* to a *long int* value. The **strtoll**() function converts the string in *nptr* to a *long long int* value. The **strtoimax**() function converts the string in *nptr* to an *intmax_t* value. The **strtoq**() function converts the string in *nptr* to a *quad_t* value. The conversion is done according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace(3)) followed by a single optional '+' or '-' sign. If *base* is zero or 16, the string may then include a '0x' prefix, and the number will be read in base 16; otherwise, a zero *base* is taken as 10 (decimal) unless the next character is '0', in which case it is taken as 8 (octal).

The remainder of the string is converted to a *long* value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter 'A' in either upper or lower case represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If *endptr* is non nil, **strtol**() stores the address of the first invalid character in ∗*endptr*. If there were no digits at all, however, **strtol**() stores the original value of *nptr* in ∗*endptr*. (Thus, if ∗*nptr* is not '\0' but ∗∗*endptr* is '\0' on return, the entire string was valid.)

RETURN VALUES

The **strtol**() function returns the result of the conversion, unless the value would underflow or overflow. If an underflow occurs, **strtol**() returns LONG_MIN, **strtoll**() returns LLONG_MIN, and **strtoimax**() returns INTMAX_MIN. If an overflow occurs, **strtol**() returns LONG_MAX, **strtoll**() returns LLONG_MAX, and **strtoimax**() returns INTMAX_MAX. In these cases, *errno* is set to ERANGE.

EXAMPLES

Ensuring that a string is a valid number (i.e., in range and containing no trailing characters) requires clearing *errno* beforehand explicitly since *errno* is not changed on a successful call to **strtol**(), and the return value of **strtol**() cannot be used unambiguously to signal an error:

```
char *ep;
long lval;

...

errno = 0;
lval = strtol(buf, &ep, 10);
if (buf[0] == '\0' || *ep != '\0')
        goto not_a_number;
if (errno == ERANGE && (lval == LONG_MAX || lval == LONG_MIN))
        goto out_of_range;
```

This example will accept "12" but not "12foo" or "12\n". If trailing whitespace is acceptable, further checks must be done on **ep*; alternately, use sscanf(3).

If **strtol**() is being used instead of atoi(3), error checking is further complicated because the desired return value is an int rather than a long; however, on some architectures integers and long integers are the same size. Thus the following is necessary:

```
char *ep;
int ival;
long lval;

...

errno = 0;
lval = strtol(buf, &ep, 10);
if (buf[0] == '\0' || *ep != '\0')
        goto not_a_number;
if ((errno == ERANGE && (lval == LONG_MAX || lval == LONG_MIN)) ||
    (lval > INT_MAX || lval < INT_MIN))
        goto out_of_range;
ival = lval;
```

ERRORS

[ERANGE] The given string was out of range; the value converted has been clamped.

SEE ALSO

atof(3), atoi(3), atol(3), atoll(3), strtod(3), strtoul(3), strtoull(3), strtoumax(3)

STANDARDS

The **strtol**() function conforms to ANSI X3.159-1989 ("ANSI C89"). The **strtoll**() and **strtoimax**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

BUGS

Ignores the current locale.

NAME

strtoul, **strtoull**, **strtoumax**, **strtouq** — convert a string to an unsigned long, unsigned long long, uintmax_t or uquad_t integer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>
#include <limits.h>

unsigned long int
strtoul(const char * restrict nptr, char ** restrict endptr, int base);

unsigned long long int
strtoull(const char * restrict nptr, char ** restrict endptr, int base);

#include <inttypes.h>

uintmax_t
strtoumax(const char * restrict nptr, char ** restrict endptr, int base);

#include <sys/types.h>
#include <stdlib.h>
#include <limits.h>

u_quad_t
strtouq(const char * restrict nptr, char ** restrict endptr, int base);
```

DESCRIPTION

The **strtoul**() function converts the string in *nptr* to an *unsigned long int* value. The **strtoull**() function converts the string in *nptr* to an *unsigned long long int* value. The **strtoumax**() function converts the string in *nptr* to an *uintmax_t* value. The **strtouq**() function converts the string in *nptr* to a *u_quad_t* value. The conversion is done according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by `isspace`(3)) followed by a single optional '`+`' or '`-`' sign. If *base* is zero or 16, the string may then include a '`0x`' prefix, and the number will be read in base 16; otherwise, a zero *base* is taken as 10 (decimal) unless the next character is '`0`', in which case it is taken as 8 (octal).

The remainder of the string is converted to an *unsigned long* value in the obvious manner, stopping at the end of the string or at the first character that does not produce a valid digit in the given base. (In bases above 10, the letter '`A`' in either upper or lower case represents 10, '`B`' represents 11, and so forth, with '`Z`' representing 35.)

If *endptr* is non nil, **strtoul**() stores the address of the first invalid character in ∗*endptr*. If there were no digits at all, however, **strtoul**() stores the original value of *nptr* in ∗*endptr*. (Thus, if ∗*nptr* is not '`\0`' but ∗∗*endptr* is '`\0`' on return, the entire string was valid.)

RETURN VALUES

The **strtoul**() function returns either the result of the conversion or, if there was a leading minus sign, the negation of the result of the conversion, unless the original (non-negated) value would overflow; in the latter case, **strtoul**() returns `ULONG_MAX`, **strtoull**() returns `ULLONG_MAX`, **strtoumax**() returns `UINTMAX_MAX`, and the global variable *errno* is set to `ERANGE`.

There is no way to determine if **strtoul**() has processed a negative number (and returned an unsigned value) short of examining the string in *nptr* directly.

EXAMPLES

Ensuring that a string is a valid number (i.e., in range and containing no trailing characters) requires clearing *errno* beforehand explicitly since *errno* is not changed on a successful call to **strtoul**(), and the return value of **strtoul**() cannot be used unambiguously to signal an error:

```
char *ep;
unsigned long ulval;

...

errno = 0;
ulval = strtoul(buf, &ep, 10);
if (buf[0] == '\0' || *ep != '\0')
        goto not_a_number;
if (errno == ERANGE && ulval == ULONG_MAX)
        goto out_of_range;
```

This example will accept “12” but not “12foo” or “12\n”. If trailing whitespace is acceptable, further checks must be done on **ep*; alternately, use sscanf(3).

ERRORS

[ERANGE] The given string was out of range; the value converted has been clamped.

SEE ALSO

strtoimax(3), strtol(3), strtoll(3)

STANDARDS

The **strtoul**() function conforms to ANSI X3.159-1989 (“ANSI C89”). The **strtoull**() and **strtoumax**() functions conform to ISO/IEC 9899:1999 (“ISO C99”).

BUGS

Ignores the current locale.

NAME

strxfrm — transform a string under locale

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <string.h>

size_t
strxfrm(char * restrict dst, const char * restrict src, size_t n);
```

DESCRIPTION

The idea of **strxfrm**() is to “un-localize” a string: the function transforms *src*, storing the result in *dst*, such that strcmp(3) on transformed strings returns what strcoll(3) on the original untransformed strings would return.

SEE ALSO

bcmp(3), memcmp(3), strcasecmp(3), strcmp(3), strcoll(3)

STANDARDS

The **strxfrm**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

BUGS

Since locales are not fully implemented on NetBSD, **strxfrm**() just returns a copy of the original string.

NAME

stty, **gtty** — set and get terminal state (defunct)

LIBRARY

Compatibility Library (libcompat, −lcompat)

SYNOPSIS

```
#include <sgtty.h>

stty(int fd, struct sgttyb *buf);

gtty(int fd, struct sgttyb *buf);
```

DESCRIPTION

These interfaces are obsoleted by `ioctl`(2). They are available from the compatibility library, libcompat.

The **stty**() function sets the state of the terminal associated with *fd*. The **gtty**() function retrieves the state of the terminal associated with *fd*. To set the state of a terminal the call must have write permission.

The **stty**() call is actually `ioctl(fd, TIOCSETP, buf)`, while the **gtty**() call is `ioctl(fd, TIOCGETP, buf)`. See `ioctl`(2) and `tty`(4) for an explanation.

DIAGNOSTICS

If the call is successful 0 is returned, otherwise −1 is returned and the global variable *errno* contains the reason for the failure.

SEE ALSO

`ioctl`(2), `tty`(4)

HISTORY

The **stty**() and **gtty**() functions appeared in 4.2BSD.

NAME

swab — swap adjacent bytes

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

void
swab(const void *src, void *dst, size_t len);
```

DESCRIPTION

The function **swab**() copies *len* bytes from the location referenced by *src* to the location referenced by *dst*, swapping adjacent bytes.

The argument *len* must be even number.

SEE ALSO

bzero(3), memset(3)

HISTORY

A **swab**() function appeared in Version 7 AT&T UNIX.

NAME

swapon — add a swap device for interleaved paging/swapping

SYNOPSIS

```
#include <unistd.h>

int
swapon(const char *special);
```

DESCRIPTION

This interface is provided for compatibility only and has been obsoleted by `swapctl`**(2).**

swapon() makes the block device *special* available to the system for allocation for paging and swapping. The names of potentially available devices are known to the system and defined at system configuration time. The size of the swap area on *special* is calculated at the time the device is first made available for swapping.

RETURN VALUES

If an error has occurred, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

swapon() succeeds unless:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG] A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.

[ENOENT] The named device does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The caller is not the super-user.

[ENOTBLK] *special* is not a block device.

[EBUSY] The device specified by *special* has already been made available for swapping

[EINVAL] The device configured by *special* was not configured into the system as a swap device.

[ENXIO] The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).

[EIO] An I/O error occurred while opening the swap device.

[EFAULT] *special* points outside the process's allocated address space.

SEE ALSO

`swapctl`(2), `swapctl`(8), `swapon`(8)

HISTORY

The **swapon**() function call appeared in 4.0BSD and was removed NetBSD 1.3

BUGS

This call will be upgraded in future versions of the system.

NAME

sysconf — get configurable system variables

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

long
sysconf(int name);
```

DESCRIPTION

This interface is defined by IEEE Std 1003.1-1988 (“POSIX.1”). A far more complete interface is available using sysctl(3).

The **sysconf**() function provides a method for applications to determine the current value of a configurable system limit or option variable. The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file ⟨unistd.h⟩.

The available values are as follows:

_SC_ARG_MAX
: The maximum bytes of argument to execve(2).

_SC_ATEXIT_MAX
: The maxmimum number of functions that may be registered with atexit(3).

_SC_BARRIERS
: The version of IEEE Std 1003.1 (“POSIX.1”) and its Barriers option to which the system attempts to conform, otherwise −1.

_SC_CLOCK_SELECTION
: Return the POSIX version the implementation of the Clock Selection Option on this system conforms to, or −1 if unavailable.

_SC_CHILD_MAX
: The maximum number of simultaneous processes per user id.

_SC_CLK_TCK
: The number of clock ticks per second.

_SC_FSYNC
: Return 1 if the File Synchronization Option is available on this system, otherwise −1.

_SC_IOV_MAX
: The maximum number of *iovec* structures that a process has available for use with preadv(2), pwritev(2), readv(2), recvmsg(2), sendmsg(2) or writev(2).

_SC_JOB_CONTROL
: Return 1 if job control is available on this system, otherwise −1.

_SC_LOGIN_NAME_MAX
: Returns the size of the storage required for a login name, in bytes, including the terminating NUL.

_SC_MAPPED_FILES
: Return 1 if the Memory Mapped Files Option is available on this system, otherwise −1.

`_SC_MEMLOCK`
Return 1 if the Process Memory Locking Option is available on this system, otherwise −1.

`_SC_MEMLOCK_RANGE`
Return 1 if the Range Memory Locking Option is available on this system, otherwise −1.

`_SC_MEMORY_PROTECTION`
Return 1 if the Memory Protection Option is available on this system, otherwise −1.

`_SC_MONOTONIC_CLOCK`
Return the POSIX version the implementation of the Monotonic Clock Option on this system conforms to, or −1 if unavailable.

`_SC_NGROUPS_MAX`
The maximum number of supplemental groups.

`_SC_OPEN_MAX`
The maximum number of open files per process.

`_SC_PAGESIZE`
The size of a system page in bytes.

`_SC_READER_WRITER_LOCKS`
The version of IEEE Std 1003.1 (“POSIX.1”) and its Read-Write Locks option to which the system attempts to conform, otherwise −1.

`_SC_SEMAPHORES`
The version of IEEE Std 1003.1 (“POSIX.1”) and its Semaphores option to which the system attempts to conform, otherwise −1.

Availability of the Semaphores option depends on the `P1003_1B_SEMAPHORE` kernel option.

`_SC_SPIN_LOCKS`
The version of IEEE Std 1003.1 (“POSIX.1”) and its Spin Locks option to which the system attempts to conform, otherwise −1.

`_SC_STREAM_MAX`
The minimum maximum number of streams that a process may have open at any one time.

`_SC_SYNCHRONIZED_IO`
Return 1 if the Synchronized I/O Option is available on this system, otherwise −1.

`_SC_THREADS`
The version of IEEE Std 1003.1 (“POSIX.1”) and its Threads option to which the system attempts to conform, otherwise −1.

`_SC_TIMERS`
The version of IEEE Std 1003.1 (“POSIX.1”) and its Timers option to which the system attempts to conform, otherwise −1.

`_SC_TZNAME_MAX`
The minimum maximum number of types supported for the name of a timezone.

`_SC_SAVED_IDS`
Returns 1 if saved set-group and saved set-user ID is available, otherwise −1.

`_SC_VERSION`
The version of ISO/IEC 9945 (POSIX 1003.1) with which the system attempts to comply.

_SC_XOPEN_SHM
: Return 1 if the X/Open Portability Guide Issue 4, Version 2 ("XPG4.2") Shared Memory option is available on this system, otherwise −1.

  Availability of the Shared Memory option depends on the SYSVSHM kernel option.

_SC_BC_BASE_MAX
: The maximum ibase/obase values in the bc(1) utility.

_SC_BC_DIM_MAX
: The maximum array size in the bc(1) utility.

_SC_BC_SCALE_MAX
: The maximum scale value in the bc(1) utility.

_SC_BC_STRING_MAX
: The maximum string length in the bc(1) utility.

_SC_COLL_WEIGHTS_MAX
: The maximum number of weights that can be assigned to any entry of the LC_COLLATE order keyword in the locale definition file.

_SC_EXPR_NEST_MAX
: The maximum number of expressions that can be nested within parenthesis by the expr(1) utility.

_SC_LINE_MAX
: The maximum length in bytes of a text-processing utility's input line.

_SC_RE_DUP_MAX
: The maximum number of repeated occurrences of a regular expression permitted when using interval notation.

_SC_2_VERSION
: The version of POSIX 1003.2 with which the system attempts to comply.

_SC_2_C_BIND
: Return 1 if the system's C-language development facilities support the C-Language Bindings Option, otherwise −1.

_SC_2_C_DEV
: Return 1 if the system supports the C-Language Development Utilities Option, otherwise −1.

_SC_2_CHAR_TERM
: Return 1 if the system supports at least one terminal type capable of all operations described in POSIX 1003.2, otherwise −1.

_SC_2_FORT_DEV
: Return 1 if the system supports the FORTRAN Development Utilities Option, otherwise −1.

_SC_2_FORT_RUN
: Return 1 if the system supports the FORTRAN Runtime Utilities Option, otherwise −1.

_SC_2_LOCALEDEF
: Return 1 if the system supports the creation of locales, otherwise −1.

_SC_2_SW_DEV
: Return 1 if the system supports the Software Development Utilities Option, otherwise −1.

_SC_2_UPE
: Return 1 if the system supports the User Portability Utilities Option, otherwise −1.

_SC_GETGR_R_SIZE_MAX
The minimum size of the *buffer* passed to getgrgid_r(3) and getgrnam_r(3).

_SC_GETPW_R_SIZE_MAX
The minimum size of the *buffer* passed to getpwnam_r(3) and getpwuid_r(3).

_SC_NPROCESSORS_CONF
The number of processors configured.

_SC_NPROCESSORS_ONLN
The number of processors online (capable of running processes).

RETURN VALUES

If the call to **sysconf** is not successful, −1 is returned and *errno* is set appropriately. Otherwise, if the variable is associated with functionality that is not supported, −1 is returned and *errno* is not modified. Otherwise, the current variable value is returned.

ERRORS

The **sysconf**() function may fail and set *errno* for any of the errors specified for the library functions sysctl(3). In addition, the following error may be reported:

[EINVAL] The value of the *name* argument is invalid.

SEE ALSO

sysctl(3)

STANDARDS

The **sysconf**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”). The constants _SC_NPROCESSORS_CONF and _SC_NPROCESSORS_ONLN are not part of the standard, but are provided by many systems.

HISTORY

The **sysconf** function first appeared in 4.4BSD.

BUGS

The value for _SC_STREAM_MAX is a minimum maximum, and required to be the same as ANSI C’s FOPEN_MAX, so the returned value is a ridiculously small and misleading number.

NAME

sysctl, **sysctlbyname**, **sysctlgetmibinfo**, **sysctlnametomib** — get or set system information

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/param.h>
#include <sys/sysctl.h>

int
sysctl(const int *name, u_int namelen, void *oldp, size_t *oldlenp,
    const void *newp, size_t newlen);

int
sysctlbyname(const char *sname, void *oldp, size_t *oldlenp, void *newp,
    size_t newlen);

int
sysctlgetmibinfo(const char *sname, int *name, u_int *namelenp, char *cname,
    size_t *csz, struct sysctlnode **rnode, int v);

int
sysctlnametomib(const char *sname, int *name, size_t *namelenp);
```

DESCRIPTION

The **sysctl** function retrieves system information and allows processes with appropriate privileges to set system information. The information available from **sysctl** consists of integers, strings, and tables. Information may be retrieved and set from the command interface using the sysctl(8) utility.

Unless explicitly noted below, **sysctl** returns a consistent snapshot of the data requested. Consistency is obtained by locking the destination buffer into memory so that the data may be copied out without blocking. Calls to **sysctl** are serialized to avoid deadlock.

The state is described using a ‘‘Management Information Base’’ (MIB) style name, listed in *name*, which is a *namelen* length array of integers.

The **sysctlbyname**() function accepts a string representation of a MIB entry and internally maps it to the appropriate numeric MIB representation. Its semantics are otherwise no different from **sysctl**().

The information is copied into the buffer specified by *oldp*. The size of the buffer is given by the location specified by *oldlenp* before the call, and that location gives the amount of data copied after a successful call. If the amount of data available is greater than the size of the buffer supplied, the call supplies as much data as fits in the buffer provided and returns with the error code ENOMEM. If the old value is not desired, *oldp* and *oldlenp* should be set to NULL.

The size of the available data can be determined by calling **sysctl** with a NULL parameter for *oldp*. The size of the available data will be returned in the location pointed to by *oldlenp*. For some operations, the amount of space may change often. For these operations, the system attempts to round up so that the returned size is large enough for a call to return the data shortly thereafter.

To set a new value, *newp* is set to point to a buffer of length *newlen* from which the requested value is to be taken. If a new value is not to be set, *newp* should be set to NULL and *newlen* set to 0.

The **sysctlnametomib**() function can be used to map the string representation of a MIB entry to the numeric version. The *name* argument should point to an array of integers large enough to hold the MIB, and

namelenp should indicate the number of integer slots available. Following a successful translation, the size_t indicated by *namelenp* will be changed to show the number of slots consumed.

The **sysctlgetmibinfo**() function performs name translation similar to **sysctlnametomib**(), but also canonicalizes the name (or returns the first erroneous token from the string being parsed) into the space indicated by *cname* and *csz*. *csz* should indicate the size of the buffer pointed to by *cname* and on return, will indicate the size of the returned string including the trailing 'nul' character.

The *rnode* and *v* arguments to **sysctlgetmibinfo**() are used to provide a tree for it to parse into, and to get back either a pointer to, or a copy of, the terminal node. If *rnode* is NULL, **sysctlgetmibinfo**() uses its own internal tree for parsing, and checks it against the kernel at each call, to make sure that the name-to-number mapping is kept up to date. The *v* argument is ignored in this case. If *rnode* is not NULL but the pointer it references is, on a successful return, *rnode* will be adjusted to point to a copy of the terminal node. The *v* argument indicates which version of the **sysctl** node structure the caller wants. The application must later **free**() this copy. If neither *rnode* nor the pointer it references are NULL, the pointer is used as the address of a tree over which the parsing is done. In this last case, the tree is not checked against the kernel, no refreshing of the mappings is performed, and the value given by *v* must agree with the version indicated by the tree. It is recommended that applications always use SYSCTL_VERSION as the value for *v*, as defined in the include file sys/sysctl.h.

The numeric and text names of sysctl variables are described in sysctl(7). The numeric names are defined as preprocessor macros. The top level names are defined with a CTL_ prefix in ⟨sys/sysctl.h⟩. The next and subsequent levels down have different prefixes for each subtree.

For example, the following retrieves the maximum number of processes allowed in the system - the kern.maxproc variable:

```
int mib[2], maxproc;
size_t len;

mib[0] = CTL_KERN;
mib[1] = KERN_MAXPROC;
len = sizeof(maxproc);
sysctl(mib, 2, &maxproc, &len, NULL, 0);
```

To retrieve the standard search path for the system utilities - user.cs_path:

```
int mib[2];
size_t len;
char *p;

mib[0] = CTL_USER;
mib[1] = USER_CS_PATH;
sysctl(mib, 2, NULL, &len, NULL, 0);
p = malloc(len);
sysctl(mib, 2, p, &len, NULL, 0);
```

DYNAMIC OPERATIONS

Several meta-identifiers are provided to perform operations on the **sysctl** tree itself, or support alternate means of accessing the data instrumented by the **sysctl** tree.

Name	Description
CTL_QUERY	Retrieve a mapping of names to numbers below a given node
CTL_CREATE	Create a new node

CTL_CREATESYM	Create a new node by its kernel symbol
CTL_DESTROY	Destroy a node
CTL_DESCRIBE	Retrieve node descriptions

The core interface to all of these meta-functions is the structure that the kernel uses to describe the tree internally, as defined in ⟨sys/sysctl.h⟩ as:

```
struct sysctlnode {
        uint32_t sysctl_flags;          /* flags and type */
        int32_t sysctl_num;             /* mib number */
        char sysctl_name[SYSCTL_NAMELEN]; /* node name */
        uint32_t sysctl_ver;        /* node's version vs. rest of tree */
        uint32_t __rsvd;
        union {
                struct {
                        uint32_t suc_csize; /* size of child node array */
                        uint32_t suc_clen; /* number of valid children */
                        struct sysctlnode* suc_child; /* array of child nodes */
                } scu_child;
                struct {
                        void *sud_data; /* pointer to external data */
                        size_t sud_offset; /* offset to data */
                } scu_data;
                int32_t scu_alias;      /* node this node refers to */
                int32_t scu_idata;      /* immediate "int" data */
                u_quad_t scu_qdata;     /* immediate "u_quad_t" data */
        } sysctl_un;
        size_t _sysctl_size;            /* size of instrumented data */
        sysctlfn _sysctl_func;          /* access helper function */
        struct sysctlnode *sysctl_parent; /* parent of this node */
        const char *sysctl_desc;        /* description of node */
};

#define sysctl_csize    sysctl_un.scu_child.suc_csize
#define sysctl_clen     sysctl_un.scu_child.suc_clen
#define sysctl_child    sysctl_un.scu_child.suc_child
#define sysctl_data     sysctl_un.scu_data.sud_data
#define sysctl_offset   sysctl_un.scu_data.sud_offset
#define sysctl_alias    sysctl_un.scu_alias
#define sysctl_idata    sysctl_un.scu_idata
#define sysctl_qdata    sysctl_un.scu_qdata
```

Querying the tree to discover the name to number mapping permits dynamic discovery of all the data that the tree currently has instrumented. For example, to discover all the nodes below the CTL_VFS node:

```
struct sysctlnode query, vfs[128];
int mib[2];
size_t len;

mib[0] = CTL_VFS;
mib[1] = CTL_QUERY;
memset(&query, 0, sizeof(query));
query.sysctl_flags = SYSCTL_VERSION;
len = sizeof(vfs);
```

```
sysctl(mib, 2, &vfs[0], &len, &query, sizeof(query));
```

Note that a reference to an empty node with *sysctl_flags* set to SYSCTL_VERSION is passed to sysctl in order to indicate the version that the program is using. All dynamic operations passing nodes into sysctl require that the version be explicitly specified.

Creation and destruction of nodes works by constructing part of a new node description (or a description of the existing node) and invoking CTL_CREATE (or CTL_CREATESYM) or CTL_DESTROY at the parent of the new node, with a pointer to the new node passed via the *new* and *newlen* arguments. If valid values for *old* and *oldlenp* are passed, a copy of the new node once in the tree will be returned. If the create operation fails because a node with the same name or MIB number exists, a copy of the conflicting node will be returned.

The minimum requirements for creating a node are setting the *sysctl_flags* to indicate the new node's type, *sysctl_num* to either the new node's number (or CTL_CREATE or CTL_CREATESYM if a dynamically allocated MIB number is acceptable), *sysctl_size* to the size of the data to be instrumented (which must agree with the given type), and *sysctl_name* must be set to the new node's name. Nodes that are not of type "node" must also have some description of the data to be instrumented, which will vary depending on what is to be instrumented.

If existing kernel data is to be covered by this new node, its address should be given in *sysctl_data* or, if CTL_CREATESYM is used, *sysctl_data* should be set to a string containing its name from the kernel's symbol table. If new data is to be instrumented and an initial value is available, the new integer or quad type data should be placed into either *sysctl_idata* or *sysctl_qdata*, respectively, along with the SYSCTL_IMMEDIATE flag being set, or *sysctl_data* should be set to point to a copy of the new data, and the SYSCTL_OWNDATA flag must be set. This latter method is the only way that new string and struct type nodes can be initialized. Invalid kernel addresses are accepted, but any attempt to access those nodes will return an error.

The *sysctl_csize*, *sysctl_clen*, *sysctl_child*, *sysctl_parent*, and *sysctl_alias* members are used by the kernel to link the tree together and must be NULL or 0. Nodes created in this manner cannot have helper functions, so *sysctl_func* must also be NULL. If the *sysctl_ver* member is non-zero, it must match either the version of the parent or the version at the root of the MIB or an error is returned. This can be used to ensure that nodes are only added or removed from a known state of the tree. Note: It may not be possible to determine the version at the root of the tree.

This example creates a new subtree and adds a node to it that controls the *audiodebug* kernel variable, thereby making it tunable at at any time, without needing to use ddb(4) or kvm(3) to alter the kernel's memory directly.

```
struct sysctlnode node;
int mib[2];
size_t len;

mib[0] = CTL_CREATE;          /* create at top-level */
len = sizeof(node);
memset(&node, 0, len);
node.sysctl_flags = SYSCTL_VERSION|CTLFLAG_READWRITE|CTLTYPE_NODE;
snprintf(node.sysctl_name, sizeof(node.sysctl_name), "local");
node.sysctl_num = CTL_CREATE; /* request dynamic MIB number */
sysctl(&mib[0], 1, &node, &len, &node, len);

mib[0] = node.sysctl_num;     /* use new MIB number */
mib[1] = CTL_CREATESYM;              /* create at second level */
len = sizeof(node);
```

```
memset(&node, 0, len);
node.sysctl_flags = SYSCTL_VERSION|CTLFLAG_READWRITE|CTLTYPE_INT;
snprintf(node.sysctl_name, sizeof(node.sysctl_name), "audiodebug");
node.sysctl_num = CTL_CREATE;
node.sysctl_data = "audiodebug"; /* kernel symbol to be used */
sysctl(&mib[0], 2, NULL, NULL, &node, len);
```

The process for deleting nodes is similar, but less data needs to be supplied. Only the *sysctl_num* field needs to be filled in; almost all other fields must be left blank. The *sysctl_name* and/or *sysctl_ver* fields can be filled in with the name and version of the existing node as additional checks on what will be deleted. If all the given data fail to match any node, nothing will be deleted. If valid values for *old* and *oldlenp* are supplied and a node is deleted, a copy of what was in the MIB tree will be returned.

This sample code shows the deletion of the two nodes created in the above example:

```
int mib[2];

len = sizeof(node);
memset(&node, 0, len);
node.sysctl_flags = SYSCTL_VERSION;

mib[0] = 3214;                /* assumed number for "local" */
mib[1] = CTL_DESTROY;
node.sysctl_num = 3215;               /* assumed number for "audiodebug" */
sysctl(&mib[0], 2, NULL, NULL, &node, len);

mib[0] = CTL_DESTROY;
node.sysctl_num = 3214;               /* now deleting "local" */
sysctl(&mib[0], 1, NULL, NULL, &node, len);
```

Descriptions of each of the nodes can also be retrieved, if they are available. Descriptions can be retrieved in bulk at each level or on a per-node basis. The layout of the buffer into which the descriptions are returned is a series of variable length structures, each of which describes its own size. The length indicated includes the terminating ‘nul’ character. Nodes that have no description or where the description is not available are indicated by an empty string. The *descr_ver* will match the *sysctl_ver* value for a given node, so that descriptions for nodes whose number have been recycled can be detected and ignored or discarded.

```
struct sysctldesc {
        int32_t         descr_num;      /* mib number of node */
        uint32_t        descr_ver;      /* version of node */
        uint32_t        descr_len;      /* length of description string */
        char            descr_str[1];   /* not really 1...see above */
};
```

The **NEXT_DESCR**() macro can be used to skip to the next description in the retrieved list.

```
struct sysctlnode desc;
struct sysctldesc *d;
char buf[1024];
int mib[2];
size_t len;

/* retrieve kern-level descriptions */
mib[0] = CTL_KERN;
mib[1] = CTL_DESCRIBE;
```

```
d = (struct sysctldesc *)&buf[0];
len = sizeof(buf);
sysctl(mib, 2, d, &len, NULL, 0);
while ((caddr_t)d < (caddr_t)&buf[len]) {
       printf("node %d: %.*s\n", d->descr_num, d->descr_len,
           d->descr_str);
       d = NEXT_DESCR(d);
}

/* retrieve description for kern.securelevel */
memset(&desc, 0, sizeof(desc));
desc.sysctl_flags = SYSCTL_VERSION;
desc.sysctl_num = KERN_SECURELEVEL;
d = (struct sysctldesc *)&buf[0];
len = sizeof(buf);
sysctl(mib, 2, d, &len, &desc, sizeof(desc));
printf("kern.securelevel: %.*s\n", d->descr_len, d->descr_str);
```

Descriptions can also be set as follows, subject to the following rules:

- The kernel securelevel is at zero or lower
- The caller has super-user privileges
- The node does not currently have a description
- The node is not marked as “permanent”

```
struct sysctlnode desc;
int mib[2];

/* presuming the given top-level node was just added... */
mib[0] = 3214; /* mib numbers taken from previous examples */
mib[1] = CTL_DESCRIBE;
memset(&desc, 0, sizeof(desc));
desc.sysctl_flags = SYSCTL_VERSION;
desc.sysctl_num = 3215;
desc.sysctl_desc = "audio debug control knob";
sysctl(mib, 2, NULL, NULL, &desc, sizeof(desc));
```

Upon successfully setting a description, the new description will be returned in the space indicated by the *oldp* and *oldlenp* arguments.

The *sysctl_flags* field in the struct sysctlnode contains the sysctl version, node type information, and a number of flags. The macros **SYSCTL_VERS**(), **SYSCTL_TYPE**(), and **SYSCTL_FLAGS**() can be used to access the different fields. Valid flags are:

Name	**Description**
CTLFLAG_READONLY	Node is read-only
CTLFLAG_READONLY1	Node becomes read-only at securelevel 1
CTLFLAG_READONLY2	Node becomes read-only at securelevel 2
CTLFLAG_READWRITE	Node is writable by the superuser
CTLFLAG_ANYWRITE	Node is writable by anyone
CTLFLAG_PRIVATE	Node is readable only by the superuser
CTLFLAG_PERMANENT	Node cannot be removed (cannot be set by processes)

CTLFLAG_OWNDATA	Node owns data and does not instrument existing data
CTLFLAG_IMMEDIATE	Node contains instrumented data and does not instrument existing data
CTLFLAG_HEX	Node's contents should be displayed in a hexadecimal form
CTLFLAG_ROOT	Node is the root of a tree (cannot be set at any time)
CTLFLAG_ANYNUMBER	Node matches any MIB number (cannot be set by processes)
CTLFLAG_HIDDEN	Node not displayed by default
CTLFLAG_ALIAS	Node refers to a sibling node (cannot be set by processes)
CTLFLAG_OWNDESC	Node owns its own description string space

RETURN VALUES

If the call to **sysctl** is successful, the number of bytes copied out is returned. Otherwise −1 is returned and *errno* is set appropriately.

FILES

⟨sys/sysctl.h⟩	definitions for top level identifiers, second level kernel and hardware identifiers, and user level identifiers
⟨sys/socket.h⟩	definitions for second level network identifiers
⟨sys/gmon.h⟩	definitions for third level profiling identifiers
⟨uvm/uvm_param.h⟩	definitions for second level virtual memory identifiers
⟨netinet/in.h⟩	definitions for third level IPv4/v6 identifiers and fourth level IPv4/v6 identifiers
⟨netinet/icmp_var.h⟩	definitions for fourth level ICMP identifiers
⟨netinet/icmp6.h⟩	definitions for fourth level ICMPv6 identifiers
⟨netinet/tcp_var.h⟩	definitions for fourth level TCP identifiers
⟨netinet/udp_var.h⟩	definitions for fourth level UDP identifiers
⟨netinet6/udp6_var.h⟩	definitions for fourth level IPv6 UDP identifiers
⟨netinet6/ipsec.h⟩	definitions for fourth level IPsec identifiers
⟨netkey/key_var.h⟩	definitions for third level PF_KEY identifiers
⟨machine/cpu.h⟩	definitions for second level machdep identifiers

ERRORS

The following errors may be reported:

[EFAULT]	The buffer *name*, *oldp*, *newp*, or length pointer *oldlenp* contains an invalid address, or the requested value is temporarily unavailable.
[EINVAL]	The *name* array is zero or greater than CTL_MAXNAME.
[EINVAL]	A non-null *newp* is given and its specified length in *newlen* is too large or too small, or the given value is not acceptable for the given node.
[EISDIR]	The *name* array specifies an intermediate rather than terminal name.
[ENOENT]	The *name* array specifies a node that does not exist in the tree.
[ENOENT]	An attempt was made to destroy a node that does not exist, or to create or destroy a node below a node that does not exist.
[ENOMEM]	The length pointed to by *oldlenp* is too short to hold the requested value.
[ENOTDIR]	The *name* array specifies a node below a node that addresses data.
[ENOTEMPTY]	An attempt was made to destroy a node that still has children.
[EOPNOTSUPP]	The *name* array specifies a value that is unknown or a meta-operation was attempted that the requested node does not support.

[EPERM]	An attempt is made to set a read-only value.
[EPERM]	A process without appropriate privilege attempts to set a value or to create or destroy a node.
[EPERM]	An attempt to change a value protected by the current kernel security level is made.

SEE ALSO

`sysctl`(7), `sysctl`(8)

HISTORY

The **`sysctl`** function first appeared in 4.4BSD.

NAME

sysexits — preferable exit codes for programs

SYNOPSIS

```
#include <sysexits.h>
```

DESCRIPTION

It is not a good practice to call exit(3) with arbitrary values to indicate a failure condition when ending a program. Instead, the pre-defined exit codes from **sysexits** should be used, so the caller of the process can get a rough estimation about the failure class without looking up the source code.

The successful exit is always indicated by a status of 0, or EX_OK. Error numbers begin at EX__BASE to reduce the possibility of clashing with other exit statuses that random programs may already return. The meaning of the codes is approximately as follows:

EX_USAGE (64)	The command was used incorrectly, e.g., with the wrong number of arguments, a bad flag, a bad syntax in a parameter, or whatever.
EX_DATAERR (65)	The input data was incorrect in some way. This should only be used for user's data and not system files.
EX_NOINPUT (66)	An input file (not a system file) did not exist or was not readable. This could also include errors like "No message" to a mailer (if it cared to catch it).
EX_NOUSER (67)	The user specified did not exist. This might be used for mail addresses or remote logins.
EX_NOHOST (68)	The host specified did not exist. This is used in mail addresses or network requests.
EX_UNAVAILABLE (69)	A service is unavailable. This can occur if a support program or file does not exist. This can also be used as a catchall message when something you wanted to do does not work, but you do not know why.
EX_SOFTWARE (70)	An internal software error has been detected. This should be limited to non-operating system related errors as possible.
EX_OSERR (71)	An operating system error has been detected. This is intended to be used for such things as "cannot fork", "cannot create pipe", or the like. It includes things like getuid returning a user that does not exist in the passwd file.
EX_OSFILE (72)	Some system file (e.g., /etc/passwd, /var/run/utmp, etc.) does not exist, cannot be opened, or has some sort of error (e.g., syntax error).
EX_CANTCREAT (73)	A (user specified) output file cannot be created.
EX_IOERR (74)	An error occurred while doing I/O on some file.
EX_TEMPFAIL (75)	Temporary failure, indicating something that is not really an error. In sendmail, this means that a mailer (e.g.) could not create a connection, and the request should be reattempted later.
EX_PROTOCOL (76)	The remote system returned something that was "not possible" during a protocol exchange.
EX_NOPERM (77)	You did not have sufficient permission to perform the operation. This is not intended for file system problems, which should use EX_NOINPUT or EX_CANTCREAT, but rather for higher level permissions.

`EX_CONFIG` (78) Something was found in an unconfigured or misconfigured state.

The numerical values corresponding to the symbolical ones are given in parenthesis for easy reference.

SEE ALSO

`err`(3), `exit`(3)

HISTORY

The **`sysexits`** file appeared somewhere after 4.3BSD. The **`sysexits`** man page appeared in NetBSD 4.0.

AUTHORS

This manual page was written by Jörg Wunsch after the comments in <`sysexits.h`>.

BUGS

The choice of an appropriate exit value is often ambiguous.

NAME

syslog, **syslog_r**, **vsyslog**, **vsyslog_r**, **openlog**, **openlog_r**, **closelog**, **closelog_r**, **setlogmask**, **setlogmask_r** — control system log

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <syslog.h>

void
syslog(int priority, const char *message, ...);

void
syslog_r(int priority, struct syslog_data *data, const char *message, ...);

void
openlog(const char *ident, int logopt, int facility);

void
openlog_r(const char *ident, int logopt, int facility,
    struct syslog_data *data);

void
closelog(void);

void
closelog_r(struct syslog_data *data);

int
setlogmask(int maskpri);

int
setlogmask_r(int maskpri, struct syslog_data *data);

#include <stdarg.h>

void
vsyslog(int priority, const char *message, va_list args);

void
vsyslog_r(int priority, struct syslog_data *data, const char *message,
    va_list args);


struct syslog_data {
        int             log_file;
        int             connected;
        int             opened;
        int             log_stat;
        const char     *log_tag;
        int             log_fac;
        int             log_mask;
};

#define SYSLOG_DATA_INIT { \
    .log_file = -1, \
    .log_fac = LOG_USER, \
```

```
    .log_mask = 0xff, \
}
```

DESCRIPTION

The **syslog**() function writes *message* to the system message logger. The message is then written to the system console, log files, logged-in users, or forwarded to other machines as appropriate (See syslogd(8)).

The message is identical to a printf(3) format string, except that '%m' is replaced by the current error message. (As denoted by the global variable *errno*; see strerror(3).) A trailing newline is added if none is present. The **syslog_r**() function is a multithread-safe version of the **syslog**() function. It takes a pointer to a *syslog_data* structure which is used to store information. This parameter must be initialized before **syslog_r**() is called. The SYSLOG_DATA_INIT constant is used for this purpose. The *syslog_data* structure is composed of the following elements:

log_file	contains the file descriptor of the file where the message is logged
connected	indicates if connect has been done
opened	indicates if **openlog_r**() has been called
log_stat	status bits, set by **openlog_r**()
log_tag	string to tag the entry with
log_fac	facility code
log_mask	mask of priorities to be logged

The **vsyslog**() function is an alternative form in which the arguments have already been captured using the variable-length argument facilities of varargs(3).

The message is tagged with *priority*. Priorities are encoded as a *facility* and a *level*. The facility describes the part of the system generating the message. The level is selected from the following *ordered* (high to low) list:

LOG_EMERG	A panic condition. This is normally broadcast to all users.
LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, e.g., hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but should possibly be handled specially.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

The **vsyslog_r**() is used the same way as **vsyslog**() except that it takes an additional pointer to a *syslog_data* structure. It is a multithread-safe version of the **vsyslog**() function described above.

The **openlog**() function provides for more specialized processing of the messages sent by **syslog**() and **vsyslog**(). The parameter *ident* is a string that will be prepended to every message. The *logopt* argument is a bit field specifying logging options, which is formed by OR'ing one or more of the following values:

LOG_CONS	If **syslog**() cannot pass the message to syslogd(8) it will attempt to write the message to the console ("/dev/console").

LOG_NDELAY Open the connection to syslogd(8) immediately. Normally the open is delayed until the first message is logged. Useful for programs that need to manage the order in which file descriptors are allocated.

LOG_PERROR Write the message to standard error output as well to the system log.

LOG_PID Log the process id with each message: useful for identifying instantiations of daemons. (This PID is placed within brackets between the ident and the message.)

The *facility* parameter encodes a default facility to be assigned to all messages that do not have an explicit facility encoded:

LOG_AUTH The authorization system: login(1), su(1), getty(8), etc.

LOG_AUTHPRIV The same as LOG_AUTH, but logged to a file readable only by selected individuals.

LOG_CRON The cron daemon: cron(8).

LOG_DAEMON System daemons, such as routed(8), that are not provided for explicitly by other facilities.

LOG_FTP The file transfer protocol daemon: ftpd(8).

LOG_KERN Messages generated by the kernel. These cannot be generated by any user processes.

LOG_LPR The line printer spooling system: lpr(1), lpc(8), lpd(8), etc.

LOG_MAIL The mail system.

LOG_NEWS The network news system.

LOG_SYSLOG Messages generated internally by syslogd(8).

LOG_USER Messages generated by random user processes. This is the default facility identifier if none is specified.

LOG_UUCP The uucp system.

LOG_LOCAL0 Reserved for local use. Similarly for LOG_LOCAL1 through LOG_LOCAL7.

The **openlog_r**() function is the multithread-safe version of the **openlog**() function. It takes an additional pointer to a *syslog_data* structure. This function must be used in conjunction with the other multithread-safe functions.

The **closelog**() function can be used to close the log file.

The **closelog_r**() does the same thing as closelog(3) but in a multithread-safe way and takes an additional pointer to a *syslog_data* structure.

The **setlogmask**() function sets the log priority mask to *maskpri* and returns the previous mask. Calls to **syslog**() with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro **LOG_MASK**(*pri*); the mask for all priorities up to and including *toppri* is given by the macro **LOG_UPTO**(*toppri*). The default allows all priorities to be logged.

The **setlogmask_r**() function is the multithread-safe version of **setlogmask**(). It takes an additional pointer to a *syslog_data* structure.

RETURN VALUES

The routines **closelog**(), **closelog_r**(), **openlog**(), **openlog_r**(), **syslog**(), **syslog_r**(), **vsyslog**(), and **vsyslog_r**() return no value.

The routines **setlogmask**() and **setlogmask_r**() always return the previous log mask level.

EXAMPLES

```
syslog(LOG_ALERT, "who: internal error 23");

openlog("ftpd", LOG_PID | LOG_NDELAY, LOG_FTP);

setlogmask(LOG_UPTO(LOG_ERR));

syslog(LOG_INFO, "Connection from host %d", CallingHost);

syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %m");
```

For the multithread-safe functions:

```
struct syslog_data sdata = SYSLOG_DATA_INIT;

syslog_r(LOG_INFO|LOG_LOCAL2, &sdata, "foobar error: %m");
```

SEE ALSO

`logger`(1), `syslogd`(8)

HISTORY

These non-multithread-safe functions appeared in 4.2BSD. The multithread-safe functions appeared in OpenBSD 3.1 and then in NetBSD 4.0. The async-signal-safe functions appeared in NetBSD 4.0.

CAVEATS

It is important never to pass a string with user-supplied data as a format without using '`%s`'. An attacker can put format specifiers in the string to mangle your stack, leading to a possible security hole. This holds true even if you have built the string "by hand" using a function like **snprintf**(), as the resulting string may still contain user-supplied conversion specifiers for later interpolation by **syslog**().

Always be sure to use the proper secure idiom:

```
syslog(priority, "%s", string);
```

NAME

system — pass a command to the shell

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
system(const char *string);
```

DESCRIPTION

The **system**() function hands the argument *string* to the command interpreter sh(1). The calling process waits for the shell to finish executing the command, ignoring SIGINT and SIGQUIT, and blocking SIGCHLD.

If *string* is a NULL pointer, **system**() will return non-zero. Otherwise, **system**() returns the termination status of the shell in the format specified by waitpid(2).

RETURN VALUES

If a child process cannot be created, or the termination status of the shell cannot be obtained, **system**() returns -1 and sets *errno* to indicate the error. If execution of the shell fails, **system**() returns the termination status for a program that terminates with a call of **exit**(*127*).

SEE ALSO

sh(1), execve(2), waitpid(2), popen(3), shquote(3)

STANDARDS

The **system**() function conforms to ANSI X3.159-1989 (“ANSI C89”) and IEEE Std 1003.2-1992 (“POSIX.2”).

CAVEATS

Never supply the **system**() function with a command containing any part of an unsanitized user-supplied string. Shell meta-characters present will be honored by the sh(1) command interpreter.

NAME

tan, **tanf** — tangent function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
tan(double x);

float
tanf(float x);
```

DESCRIPTION

The **tan**() and **tanf**() functions compute the tangent of *x* (measured in radians). A large magnitude argument may yield a result with little or no significance. For a discussion of error due to roundoff, see math(3).

RETURN VALUES

The **tan**() function returns the tangent value.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tanh(3)

STANDARDS

The **tan**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

tanh, **tanhf** — hyperbolic tangent function

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
tanh(double x);

float
tanhf(float x);
```

DESCRIPTION

The **tanh**() and **tanhf**() functions compute the hyperbolic tangent of *x*. For a discussion of error due to roundoff, see math(3).

RETURN VALUES

The **tanh**() function returns the hyperbolic tangent value.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tan(3)

STANDARDS

The **tanh**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

tcgetpgrp — get foreground process group ID

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

pid_t
tcgetpgrp(int fd);
```

DESCRIPTION

The **tcgetpgrp** function returns the value of the process group ID of the foreground process group associated with the terminal device. If there is no foreground process group, **tcgetpgrp** returns an invalid process ID.

ERRORS

If an error occurs, **tcgetpgrp** returns -1 and the global variable *errno* is set to indicate the error, as follows:

[EBADF] The *fd* argument is not a valid file descriptor.

[ENOTTY] The calling process does not have a controlling terminal or the underlying terminal device represented by *fd* is not the controlling terminal.

SEE ALSO

setpgid(2), setsid(2), tcsetpgrp(3)

STANDARDS

The **tcgetpgrp** function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

tcgetsid — get session ID associated with a controlling terminal

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t
tcgetsid(int fd);
```

DESCRIPTION

The **tcgetsid** function returns the value of the session ID associated with the specified controlling terminal device. The session ID is defined as the process group ID of the session leader.

ERRORS

If an error occurs, **tcgetsid** returns -1 and the global variable *errno* is set to indicate the error, as follows:

[EBADF] The *fd* argument is not a valid file descriptor.

[ENOTTY] The calling process does not have a controlling terminal or the underlying terminal device represented by *fd* is not the controlling terminal.

SEE ALSO

getsid(2), setsid(2), tcgetpgrp(3)

STANDARDS

The **tcgetsid** function conforms to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

NAME

tcsendbreak, **tcdrain**, **tcflush**, **tcflow** — line control functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <termios.h>

int
tcdrain(int fd);

int
tcflow(int fd, int action);

int
tcflush(int fd, int action);

int
tcsendbreak(int fd, int len);
```

DESCRIPTION

The **tcdrain** function waits until all output written to the terminal referenced by *fd* has been transmitted to the terminal.

The **tcflow** function suspends transmission of data to or the reception of data from the terminal referenced by *fd* depending on the value of *action*. The value of *action* must be one of the following:

TCOOFF Suspend output.

TCOON Restart suspended output.

TCIOFF Transmit a STOP character, which is intended to cause the terminal to stop transmitting data to the system. (See the description of IXOFF in the `Input Modes` section of `termios`(4)).

TCION Transmit a START character, which is intended to cause the terminal to start transmitting data to the system. (See the description of IXOFF in the `Input Modes` section of `termios`(4)).

The **tcflush** function discards any data written to the terminal referenced by *fd* which has not been transmitted to the terminal, or any data received from the terminal but not yet read, depending on the value of *action*. The value of *action* must be one of the following:

TCIFLUSH Flush data received but not read.

TCOFLUSH Flush data written but not transmitted.

TCIOFLUSH Flush both data received but not read and data written but not transmitted.

The **tcsendbreak** function transmits a continuous stream of zero-valued bits for four-tenths of a second to the terminal referenced by *fd*. The *len* parameter is ignored in this implementation.

RETURN VALUES

Upon successful completion, all of these functions return a value of zero.

ERRORS

If any error occurs, a value of -1 is returned and the global variable *errno* is set to indicate the error, as follows:

[EBADF] The *fd* argument is not a valid file descriptor.

[EINVAL] The *action* argument is not a proper value.

[ENOTTY] The file associated with *fd* is not a terminal.

[EINTR] A signal interrupted the **tcdrain** function.

SEE ALSO

tcsetattr(3), termios(4)

STANDARDS

The **tcsendbreak**, **tcdrain**, **tcflush** and **tcflow** functions are expected to be compliant with the IEEE Std 1003.1-1988 (“POSIX.1”) specification.

NAME

cfgetispeed, **cfsetispeed**, **cfgetospeed**, **cfsetospeed**, **cfsetspeed**, **cfmakeraw**, **tcgetattr**, **tcsetattr** — manipulating the termios structure

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <termios.h>

speed_t
cfgetispeed(const struct termios *t);

int
cfsetispeed(struct termios *t, speed_t speed);

speed_t
cfgetospeed(const struct termios *t);

int
cfsetospeed(struct termios *t, speed_t speed);

int
cfsetspeed(struct termios *t, speed_t speed);

void
cfmakeraw(struct termios *t);

int
tcgetattr(int fd, struct termios *t);

int
tcsetattr(int fd, int action, const struct termios *t);
```

DESCRIPTION

The **cfmakeraw**, **tcgetattr** and **tcsetattr** functions are provided for getting and setting the termios structure.

The **cfgetispeed**, **cfsetispeed**, **cfgetospeed**, **cfsetospeed** and **cfsetspeed** functions are provided for getting and setting the baud rate values in the termios structure. The effects of the functions on the terminal as described below do not become effective, nor are all errors detected, until the **tcsetattr** function is called. Certain values for baud rates set in the termios structure and passed to **tcsetattr** have special meanings. These are discussed in the portion of the manual page that describes the **tcsetattr** function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the termios structure. The unsigned integer `speed_t` is typdef'd in the include file ⟨`termios.h`⟩. The value of the integer corresponds directly to the baud rate being represented, however, the following symbolic values are defined.

```
#define B0     0
#define B50    50
#define B75    75
#define B110   110
#define B134   134
#define B150   150
#define B200   200
```

```
#define B300   300
#define B600   600
#define B1200  1200
#define B1800  1800
#define B2400  2400
#define B4800  4800
#define B9600  9600
#define B19200 19200
#define B38400 38400
#ifndef _POSIX_SOURCE
#define EXTA   19200
#define EXTB   38400
#endif  /*_POSIX_SOURCE */
```

The **cfgetispeed** function returns the input baud rate in the termios structure referenced by *tp*.

The **cfsetispeed** function sets the input baud rate in the termios structure referenced by *tp* to *speed*.

The **cfgetospeed** function returns the output baud rate in the termios structure referenced by *tp*.

The **cfsetospeed** function sets the output baud rate in the termios structure referenced by *tp* to *speed*.

The **cfsetspeed** function sets both the input and output baud rate in the termios structure referenced by *tp* to *speed*.

Upon successful completion, the functions **cfsetispeed**, **cfsetospeed**, and **cfsetspeed** return a value of 0. Otherwise, a value of -1 is returned and the global variable *errno* is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation and the SIGTTOU signal is not sent.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The **cfmakeraw** function sets the flags stored in the termios structure (initialized by **tcgetattr**) to a state disabling all input and output processing, giving a "raw I/O path". It should be noted that there is no function to reverse this effect. This is because there are a variety of processing options that could be re-enabled and the correct method is for an application to snapshot the current terminal state using the function **tcgetattr**, setting raw mode with **cfmakeraw** and the subsequent **tcsetattr**, and then using another **tcsetattr** with the saved state to revert to the previous terminal state.

The **tcgetattr** function copies the parameters associated with the terminal referenced by *fd* to the termios structure referenced by *tp*. This function is allowed from a background process, however, the terminal attributes may be subsequently changed by a foreground process.

The **tcsetattr** function sets the parameters associated with the terminal from the termios structure referenced by *tp*. The *action* field is created by *or*'ing the following values, as specified in the include file ⟨termios.h⟩.

TCSANOW The change occurs immediately.

TCSADRAIN The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.

TCSAFLUSH The change occurs after all output written to *fd* has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.

TCSASOFT If this value is *or*'ed into the *action* value, the values of the *c_cflag*, *c_ispeed*, and *c_ospeed* fields are ignored.

The 0 baud rate is used to terminate the connection. If 0 is specified as the output speed to the function **tcsetattr**, modem control will no longer be asserted on the terminal, disconnecting the terminal.

If zero is specified as the input speed to the function **tcsetattr**, the input baud rate will be set to the same value as that specified by the output baud rate.

RETURN VALUES

If **tcsetattr** is unable to make any of the requested changes, it returns -1 and sets errno. Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions **tcgetattr** and **tcsetattr** return a value of 0. Otherwise, they return -1 and the global variable *errno* is set to indicate the error, as follows:

[EBADF] The *fd* argument to **tcgetattr** or **tcsetattr** was not a valid file descriptor.

[EINTR] The **tcsetattr** function was interrupted by a signal.

[EINVAL] The *action* argument to the **tcsetattr** function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.

[ENOTTY] The file associated with the *fd* argument to **tcgetattr** or **tcsetattr** is not a terminal.

SEE ALSO

tcsendbreak(3), termios(4)

STANDARDS

The **cfgetispeed**, **cfsetispeed**, **cfgetospeed**, **cfsetospeed**, **tcgetattr** and **tcsetattr** functions are expected to be compliant with the IEEE Std 1003.1-1988 ("POSIX.1") specification. The **cfmakeraw** and **cfsetspeed** functions, as well as the TCSASOFT option to the **tcsetattr** function are extensions to the IEEE Std 1003.1-1988 ("POSIX.1") specification.

NAME

tcsetpgrp — set foreground process group ID

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
tcsetpgrp(int fd, pid_t pgrp_id);
```

DESCRIPTION

If the process has a controlling terminal, the **tcsetpgrp** function sets the foreground process group ID associated with the terminal device to *pgrp_id*. The terminal device associated with *fd* must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. The value of *pgrp_id* must be the same as the process group ID of a process in the same session as the calling process.

Upon successful completion, **tcsetpgrp** returns a value of zero.

ERRORS

If an error occurs, **tcgetpgrp** returns -1 and the global variable *errno* is set to indicate the error, as follows:

[EBADF]	The *fd* argument is not a valid file descriptor.
[EINVAL]	An invalid value of *pgrp_id* was specified.
[ENOTTY]	The calling process does not have a controlling terminal, or the file represented by *fd* is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.
[EPERM]	The *pgrp_id* argument does not match the process group ID of a process in the same session as the calling process.

SEE ALSO

setpgid(2), setsid(2), tcgetpgrp(3)

STANDARDS

The **tcsetpgprp** function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

tgetent, **tgetnum**, **tgetflag**, **tgetstr**, **tgoto**, **tputs** — terminal independent operation routines

LIBRARY

Termcap Access Library (libtermcap, −ltermcap)

SYNOPSIS

```
#include <termcap.h>

char PC;
char *BC;
char *UP;
short ospeed;
struct tinfo *info;

int
tgetent(char *bp, const char *name);

int
tgetnum(const char *id);

int
tgetflag(const char *id);

char *
tgetstr(const char *id, char **area);

char *
tgoto(const char *cm, int destcol, int destline);

void
tputs(const char *cp, int affcnt, int (*outc)(int));

int
t_getent(struct tinfo **info, const char *name);

int
t_getnum(struct tinfo *info, const char *id);

int
t_getflag(struct tinfo *info, const char *id);

char *
t_getstr(struct tinfo *info, const char *id, char **area, size_t *limit);

char *
t_agetstr(struct tinfo *info, const char *id);

int
t_getterm(struct tinfo *info, char **area, size_t *limit);

int
t_goto(struct tinfo *info, const char *id, int destcol, int destline,
    char *buffer, size_t limit);

int
t_puts(struct tinfo *info, const char *cp, int affcnt,
    void (*outc)(char, void *), void *args);
```

```
void
t_freent(struct tinfo *info);

int
t_setinfo(struct tinfo **info, const char *entry);

#include <wchar.h>

int
t_putws(struct tinfo *info, const wchar_t *cp, int affcnt,
    void (*outc)(wchar_t, void *), void *args);
```

DESCRIPTION

These functions extract and use capabilities from a terminal capability data base, usually /usr/share/misc/termcap, the format of which is described in termcap(5). These are low level routines; see curses(3) for a higher level package.

The **tgetent**() function extracts the entry for terminal *name* into the buffer at *bp*. The *bp* argument should be a character buffer of size 1024 and must be retained through all subsequent calls to **tgetnum**(), **tgetflag**(), and **tgetstr**(). The **tgetent**() function returns −1 if none of the **termcap** data base files could be opened, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, the value does not contain the ZZ capability (see *NOTES* for a description of this capability), and the terminal type *name* is the same as the environment string TERM, the TERMCAP string is used instead of reading a **termcap** file. If the value does contain the ZZ capability then the TERM environment string is used to read **termcap**, if the read fails for any reason the value of TERMCAP will be used despite it containing ZZ. If TERMCAP does begin with a slash, the string is used as a path name of the **termcap** file to search. If TERMCAP does not begin with a slash and *name* is different from TERM, **tgetent**() searches the files $HOME/.termcap and /usr/share/misc/termcap, in that order, unless the environment variable TERMPATH exists, in which case it specifies a list of file pathnames (separated by spaces or colons) to be searched instead. Whenever multiple files are searched and a **tc** field occurs in the requested entry, the entry it names must be found in the same file or one of the succeeding files. This can speed up entry into programs that call **tgetent**(), as well as help debug new terminal descriptions or make one for your terminal if you can't write the file /usr/share/misc/termcap.

The **tgetnum**() function gets the numeric value of capability *id*, returning −1 if it is not given for the terminal. The **tgetflag**() function returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. The **tgetstr**() function returns the string value of the capability *id*; if *area* does not point to NULL and does not point to a pointer to NULL, it copies the string value into the buffer pointed to by **area*, and advances the **area* pointer past the copy of the string. It decodes the abbreviations for this field described in termcap(5), except for cursor addressing and padding information. The **tgetstr**() function returns NULL if the capability was not found.

The **tgoto**() function returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables *UP* (from the **up** capability) and *BC* (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. (Programs which call **tgoto**() should be sure to turn off the XTABS bit(s), since **tgoto**() may now output a tab. Note that programs using termcap should in general turn off XTABS anyway since some terminals use control-I for other functions, such as nondestructive space.) If a **%** sequence is given which is not understood, then **tgoto**() returns (OOPS).

The **tputs**() function decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by stty(3). The external variable *PC* should contain a pad character to be used (from the **pc** capability) if a null (**^@**) is inappropriate.

The **t_getent**() function operates in a similar manner to the **tgetent**() function excepting that the *info* argument is a pointer to a pointer of the opaque type *tinfo*. If the call to **t_getent**() succeeds then the argument *info* will be updated with the address of an object that contains the termcap entry. This pointer can then be passed to calls of **t_getnum**(), **t_getflag**() and **t_getstr**(). When the information pointed to by *info* is no longer required any storage associated with the object can be released by calling **t_freent**().

The functions **t_getnum**() and **t_getflag**() operate in the same manner as **tgetnum**() and **tgetflag**() with the exception that the pointer to the termcap object is passed along with the id of the capability required.

The function **t_getstr**() performs the same function as **tgetstr**() but has a *limit* parameter that gives the number of characters that can be inserted in to the array pointed to by *area*. The *limit* argument is updated by the **t_getstr**() call to give the number of characters that remain available in *area*. If the t_getstr call fails then NULL will be returned and errno set to indicate the failure, ENOENT indicates there was no termcap entry for the given *id*, E2BIG indicates the retrieved entry would have overflowed *area*. If t_getstr is called with *area* being NULL then the size required to hold the capability string will be returned in *limit* so the caller can allocate enough storage to hold the capability.

The function **t_agetstr**() performs the same function as **t_getstr**() except it handles memory allocation automatically. The memory that **t_agetstr**() allocates will be freed when **t_freent**() is called.

The function **t_getterm**() returns a copy of the termcap name string of the termcap entry associated with *info* in the buffer pointed to by *area*. **t_getterm**() returns 0 on success and −1 on error. On error errno will be set to EINVAL if the termcap entry in *info* is malformed or E2BIG if the size of the name exceeds the size specified by *limit*. If *area* is NULL then the size required to hold the terminal name will be returned in *limit* allowing sufficient storage to be allocated. If *limit* is NULL then no bounds checking will be performed.

The **t_goto**() function is the same as the **tgoto**() function excepting that the capabilities for **up** and **bc** are extracted from the *info* object and that the string formed by **t_goto**() is placed in the *buffer* argument, the number of characters allowed to be placed in *buffer* is controlled by *limit*. If the expansion performed by **t_goto**() would exceed the space in *buffer* then **t_goto**() will return −1 and set errno to E2BIG. The function **t_puts**() is similar to the **tputs**() function excepting that *info* holds a pointer to the termcap object that was returned by a previous **t_getent**() call, this object will be used to retrieve the **pc** attribute for the terminal. The function **t_putws**() is similar to **t_puts**() but it operates on a string of wide characters. The *outc* function is a pointer to a function that will be called by **t_puts**() to output each character in the *cp* string. The *outc* function will be called with two parameters. The first is the character to be printed and the second is an optional argument that was passed to **t_puts**() in the *args* argument. The interpretation of the contents of *args* is dependent solely on the implementation of *outc*.

The **t_setinfo**() function allows the termcap entry contained in the *entry* string to be inserted into the *info* structure. Memory sufficient to hold the contents of *entry* is automatically allocated. This allows the programmer to provide a fail over terminal capability string if fetching the termcap entry from the termcap database fails. The format of the string *entry* is assumed to be a valid termcap entry.

NOTE: A special capability of *ZZ* is added to the end of the termcap entry retrieved. The number that follows this entry is the address of the buffer allocated to hold the full termcap entry. The caller may retrieve the pointer to the extended buffer by performing a **tgetstr**() to retrieve the *ZZ* capability, the string is the output of a **printf**() %p and may be converted back to a pointer using **sscanf**() or similar. The ZZ capability is only necessary if the caller wishes to directly manipulate the termcap entry, all the termcap function calls automatically use the extended buffer to retrieve terminal capabilities.

FILES

`/usr/lib/libtermcap.a`	**`-l`** *`termcap`* library (also known as **`-l`** *`termlib`*)
`/usr/share/misc/termcap`	standard terminal capability data base
`$HOME/.termcap`	user's terminal capability data base

SEE ALSO

`ex`(1), `curses`(3), `termcap`(5)

HISTORY

The **`termcap`** t_∗() functions appeared in NetBSD 1.5. The rest of the **`termcap`** functions appeared in 4.0BSD.

NAME

textdomain − set domain for future gettext() calls

SYNOPSIS

#include <libintl.h>

char * textdomain (const char * *domainname***);**

DESCRIPTION

The **textdomain** function sets or retrieves the current message domain.

A message domain is a set of translatable *msgid* messages. Usually, every software package has its own message domain. The domain name is used to determine the message catalog where a translation is looked up; it must be a non-empty string.

The current message domain is used by the **gettext**, **ngettext** functions, and by the **dgettext**, **dcgettext**, **dngettext** and **dcngettext** functions when called with a NULL domainname argument.

If *domainname* is not NULL, the current message domain is set to *domainname*. The string the function stores internally is a copy of the *domainname* argument.

If *domainname* is NULL, the function returns the current message domain.

RETURN VALUE

If successful, the **textdomain** function returns the current message domain, after possibly changing it. The resulting string is valid until the next **textdomain** call and must not be modified or freed. If a memory allocation failure occurs, it sets **errno** to **ENOMEM** and returns NULL.

ERRORS

The following error can occur, among others:

ENOMEM

Not enough memory available.

BUGS

The return type ought to be **const char ***, but is **char *** to avoid warnings in C code predating ANSI C.

SEE ALSO

gettext(3), **ngettext**(3), **bindtextdomain**(3), **bind_textdomain_codeset**(3)

NAME

time — get time of day

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <time.h>

time_t
time(time_t *tloc);
```

DESCRIPTION

The **time**() function returns the value of time in seconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time.

A copy of the time value may be saved to the area indicated by the pointer *tloc*. If *tloc* is a NULL pointer, no value is stored.

Upon successful completion, **time**() returns the value of time. Otherwise a value of ((*time_t*) −1) is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following error codes may be set in *errno*:

[EFAULT] An argument address referenced invalid memory.

SEE ALSO

gettimeofday(2), ctime(3)

STANDARDS

The **time**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

A **time**() function appeared in Version 6 AT&T UNIX.

NAME

time2posix, **posix2time** — convert seconds since the Epoch

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <time.h>

time_t
time2posix(time_t t);

time_t
posix2time(time_t t);
```

DESCRIPTION

IEEE Std 1003.1 (“POSIX.1”) legislates that a *time_t* value of 536457599 shall correspond to

```
Wed Dec 31 23:59:59 UTC 1986
```

.

This effectively implies that POSIX *time_t*’s cannot include leap seconds and, therefore, that the system time must be adjusted as each leap occurs.

If the time package is configured with leap-second support enabled, however, no such adjustment is needed and *time_t* values continue to increase over leap events (as a true ‘seconds since...’ value). This means that these values will differ from those required by POSIX by the net number of leap seconds inserted since the Epoch.

Typically this is not a problem as the type *time_t* is intended to be (mostly) opaque — *time_t* values should only be obtained-from and passed-to functions such as `time`(3), `localtime`(3), `mktime`(3), and `difftime`(3). However, POSIX gives an arithmetic expression for directly computing a *time_t* value from a given date/time, and the same relationship is assumed by some (usually older) applications. Any programs creating/dissecting *time_t*’s using such a relationship will typically not handle intervals over leap seconds correctly.

The **time2posix**() and **posix2time**() functions are provided to address this *time_t* mismatch by converting between local *time_t* values and their POSIX equivalents. This is done by accounting for the number of time-base changes that would have taken place on a POSIX system as leap seconds were inserted or deleted. These converted values can then be used in lieu of correcting the older applications, or when communicating with POSIX-compliant systems.

time2posix() is single-valued. That is, every local *time_t* corresponds to a single POSIX *time_t*. **posix2time**() is less well-behaved: for a positive leap second hit the result is not unique, and for a negative leap second hit the corresponding POSIX *time_t* doesn’t exist so an adjacent value is returned. Both of these are good indicators of the inferiority of the POSIX representation.

The following table summarizes the relationship between a *time_t* and its conversion to, and back from, the POSIX representation over the leap second inserted at the end of June, 1993.

DATE	TIME	T	X=time2posix(T)	posix2time(X)
93/06/30	23:59:59	A+0	B+0	A+0
93/06/30	23:59:60	A+1	B+1	A+1 or A+2
93/07/01	00:00:00	A+2	B+1	A+1 or A+2
93/07/01	00:00:01	A+3	B+2	A+3

A leap second deletion would look like...

DATE	TIME	T	X=time2posix(T)	posix2time(X)
??/06/30	23:59:58	A+0	B+0	A+0
??/07/01	00:00:00	A+1	B+2	A+1
??/07/01	00:00:01	A+2	B+3	A+2

[Note: posix2time(B+1) => A+0 or A+1]

If leap-second support is not enabled, local *time_t*'s and POSIX *time_t*'s are equivalent, and both **time2posix**() and **posix2time**() degenerate to the identity function.

SEE ALSO

difftime(3), localtime(3), mktime(3), time(3)

NAME

times — process times

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/times.h>

clock_t
times(struct tms *tp);
```

DESCRIPTION

This interface is obsoleted by `getrusage`(**2**) **and** `gettimeofday`(**2**)**.**

The **times**() function returns the value of time in clock ticks since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time (UTC).

The number of clock ticks per second may be determined by calling `sysconf`(3) with the `_SC_CLK_TCK` request. It is generally (but not always) between 60 and 1024.

Note that at the common rate of 100 ticks per second on many NetBSD ports, and with a 32-bit unsigned clock_t, this value first wrapped in 1971.

The **times**() call also fills in the structure pointed to by *tp* with time-accounting information.

The *tms* structure is defined as follows:

```
typedef struct {
        clock_t tms_utime;
        clock_t tms_stime;
        clock_t tms_cutime;
        clock_t tms_cstime;
}
```

The elements of this structure are defined as follows:

tms_utime The CPU time charged for the execution of user instructions.

tms_stime The CPU time charged for execution by the system on behalf of the process.

tms_cutime The sum of the *tms_utime s* and *tms_cutime s* of the child processes.

tms_cstime The sum of the *tms_stime*s and *tms_cstime*s of the child processes.

All times are measured in clock ticks, as defined above. Note that at 100 ticks per second, and with a 32-bit unsigned clock_t, the values wrap after 497 days.

The times of a terminated child process are included in the *tms_cutime* and *tms_cstime* elements of the parent when one of the `wait`(2) functions returns the process ID of the terminated child to the parent. If an error occurs, **times**() returns the value ((clock_t)−1), and sets *errno* to indicate the error.

ERRORS

The **times**() function may fail and set the global variable *errno* for any of the errors specified for the library routines `getrusage`(2) and `gettimeofday`(2).

SEE ALSO

`time`(1), `getrusage`(2), `gettimeofday`(2), `wait`(2), `sysconf`(3)

STANDARDS

The **`times`**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

timezone — return the timezone abbreviation

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
char *
timezone(int zone, int dst);
```

DESCRIPTION

This interface is available for compatibility only and will disappear in a future software release; it is impossible to reliably map timezone's arguments to a time zone abbreviation. See ctime(**3**)**; see** tzset(**3**) **for the new definition of this interface.**

The **timezone**() function returns a pointer to a time zone abbreviation for the specified *zone* and *dst* values. *Zone* is the number of minutes west of GMT and *dst* is non-zero if daylight savings time is in effect.

SEE ALSO

ctime(3), tzset(3)

HISTORY

A **timezone**() function appeared in Version 7 AT&T UNIX.

NAME

tempnam, **tmpfile**, **tmpnam** — temporary file routines

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

FILE *
tmpfile(void);

char *
tmpnam(char *str);

char *
tempnam(const char *tmpdir, const char *prefix);
```

DESCRIPTION

The **tmpfile**() function returns a pointer to a stream associated with a file descriptor returned by the routine mkstemp(3). The created file is unlinked before **tmpfile**() returns, causing the file to be automatically deleted when the last reference to it is closed. The file is opened with the access value 'w+'.

The **tmpnam**() function returns a pointer to a file name, in the P_tmpdir directory, which did not reference an existing file at some indeterminate point in the past. P_tmpdir is defined in the include file ⟨stdio.h⟩. If the argument *s* is non-NULL, the file name is copied to the buffer it references. Otherwise, the file name is copied to a static buffer. In either case, **tmpnam**() returns a pointer to the file name.

The buffer referenced by *s* is expected to be at least L_tmpnam bytes in length. L_tmpnam is defined in the include file ⟨stdio.h⟩.

The **tempnam**() function is similar to **tmpnam**(), but provides the ability to specify the directory which will contain the temporary file and the file name prefix.

The environment variable TMPDIR (if set), the argument *tmpdir* (if non-NULL), the directory P_tmpdir, and the directory /tmp are tried, in the listed order, as directories in which to store the temporary file.

The argument *prefix*, if non-NULL, is used to specify a file name prefix, which will be the first part of the created file name. **tempnam**() allocates memory in which to store the file name; the returned pointer may be used as a subsequent argument to free(3).

RETURN VALUES

The **tmpfile**() function returns a pointer to an open file stream on success, and a NULL pointer on error.

The **tmpnam**() and **tempnam**() functions return a pointer to a file name on success, and a NULL pointer on error.

ERRORS

The **tmpfile**() function may fail and set the global variable *errno* for any of the errors specified for the library functions fdopen(3) or mkstemp(3).

The **tmpnam**() function may fail and set *errno* for any of the errors specified for the library function mktemp(3).

The **tempnam**() function may fail and set *errno* for any of the errors specified for the library functions malloc(3) or mktemp(3).

SEE ALSO

mkstemp(3), mktemp(3)

STANDARDS

The **tmpfile**() and **tmpnam**() functions conform to ANSI X3.159-1989 ("ANSI C89").

BUGS

These interfaces are provided for AT&T System V UNIX and ANSI compatibility only. The mkstemp(3) interface is strongly preferred.

SECURITY CONSIDERATIONS

There are four important problems with these interfaces (as well as with the historic mktemp(3) interface). First, there is an obvious race between file name selection and file creation and deletion: the program is typically written to call **tmpnam**(), **tempnam**(), or mktemp(3). Subsequently, the program calls open(2) or fopen(3) and erroneously opens a file (or symbolic link, or fifo or other device) that the attacker has placed in the expected file location. Hence mkstemp(3) is recommended, since it atomically creates the file.

Second, most historic implementations provide only a limited number of possible temporary file names (usually 26) before file names will start being recycled. Third, the AT&T System V UNIX implementations of these functions (and of mktemp(3)) use the access(2) system call to determine whether or not the temporary file may be created. This has obvious ramifications for setuid or setgid programs, complicating the portable use of these interfaces in such programs. Finally, there is no specification of the permissions with which the temporary files are created.

This implementation of **tmpfile**() does not have these flaws, and that of **tmpnam**() and **tempnam**() only have the first limitation, but portable software cannot depend on that. In particular, the **tmpfile**() interface should not be used in software expected to be used on other systems if there is any possibility that the user does not wish the temporary file to be publicly readable and writable.

A link-time warning will be issued if **tmpnam**() or **tempnam**() is used, and advises the use of **mkstemp**() instead.

NAME

toascii — convert a byte to 7-bit ASCII

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
toascii(int c);
```

DESCRIPTION

The **toascii**() function returns the argument with all but the lower 7 bits cleared.

RETURN VALUES

The **toascii**() function always returns a valid ASCII character. The result is a non-negative integer in the range from 0 to 127, inclusive.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), tolower(3), toupper(3), ascii(7)

STANDARDS

The **toascii**() function conforms to X/Open Portability Guide Issue 4 (“XPG4”).

NAME

tolower — upper case to lower case letter conversion

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
tolower(int c);
```

DESCRIPTION

The **tolower**() function converts an upper-case letter to the corresponding lower-case letter.

RETURN VALUES

If the argument is an upper-case letter, the **tolower**() function returns the corresponding lower-case letter if there is one; otherwise the argument is returned unchanged.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), toupper(3), ascii(7)

STANDARDS

The **tolower**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

CAVEATS

The argument to **tolower**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

toupper — lower case to upper case letter conversion

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ctype.h>

int
toupper(int c);
```

DESCRIPTION

The **toupper**() function converts a lower-case letter to the corresponding upper-case letter.

RETURN VALUES

If the argument is a lower-case letter, the **toupper**() function returns the corresponding upper-case letter if there is one; otherwise the argument is returned unchanged.

SEE ALSO

ctype(3), isalnum(3), isalpha(3), isascii(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), ascii(7)

STANDARDS

The **toupper**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

CAVEATS

The argument to **toupper**() must be EOF or representable as an *unsigned char*; otherwise, the behavior is undefined. See the **CAVEATS** section of ctype(3) for more details.

NAME

towctrans — convert a wide character with a specified map

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wctype.h>

wint_t
towctrans(wint_t wc, wctrans_t charmap);
```

DESCRIPTION

The **towctrans**() function converts a wide character *wc* with a character mapping *charmap*.

The behaviour of **towctrans**() is undefined if the **towctrans**() function is called with an invalid *charmap* (changes of LC_CTYPE category invalidate *charmap*) or invalid wide character *wc*.

The behaviour of **towctrans**() is affected by the LC_CTYPE category of the current locale.

RETURN VALUES

towctrans() returns the resulting character of the conversion.

ERRORS

No errors are defined.

SEE ALSO

iswctype(3), setlocale(3), wctrans(3), wctype(3)

STANDARDS

The **towctrans**() function conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”).

NAME

towlower — wide character case letter conversion utilities

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wctype.h>

wint_t
towlower(wint_t wc);

wint_t
towupper(wint_t wc);
```

DESCRIPTION

The **towlower**() function converts an upper-case wide character to the corresponding lower-case letter. The **towupper**() function converts an lower-case wide character to the corresponding upper-case letter.

RETURN VALUES

If the argument is an upper/lower-case letter, the **tolower**() function returns the corresponding counterpart if there is one; otherwise the argument is returned unchanged.

SEE ALSO

tolower(3), toupper(3)

STANDARDS

The functions conform to ISO/IEC 9899:1999 (“ISO C99”).

NAME

SPLAY_PROTOTYPE, **SPLAY_GENERATE**, **SPLAY_ENTRY**, **SPLAY_HEAD**, **SPLAY_INITIALIZER**, **SPLAY_ROOT**, **SPLAY_EMPTY**, **SPLAY_NEXT**, **SPLAY_MIN**, **SPLAY_MAX**, **SPLAY_FIND**, **SPLAY_LEFT**, **SPLAY_RIGHT**, **SPLAY_FOREACH**, **SPLAY_INIT**, **SPLAY_INSERT**, **SPLAY_REMOVE**, **RB_PROTOTYPE**, **RB_GENERATE**, **RB_ENTRY**, **RB_HEAD**, **RB_INITIALIZER**, **RB_ROOT**, **RB_EMPTY**, **RB_NEXT**, **RB_MIN**, **RB_MAX**, **RB_FIND**, **RB_LEFT**, **RB_RIGHT**, **RB_PARENT**, **RB_FOREACH**, **RB_INIT**, **RB_INSERT**, **RB_REMOVE** — implementations of splay and red-black trees

SYNOPSIS

#include <sys/tree.h>

SPLAY_PROTOTYPE(*NAME*, *TYPE*, *FIELD*, *CMP*);

SPLAY_GENERATE(*NAME*, *TYPE*, *FIELD*, *CMP*);

SPLAY_ENTRY(*TYPE*);

SPLAY_HEAD(*HEADNAME*, *TYPE*);

*struct TYPE **
SPLAY_INITIALIZER(*SPLAY_HEAD *head*);

SPLAY_ROOT(*SPLAY_HEAD *head*);

bool
SPLAY_EMPTY(*SPLAY_HEAD *head*);

*struct TYPE **
SPLAY_NEXT(*NAME*, *SPLAY_HEAD *head*, *struct TYPE *elm*);

*struct TYPE **
SPLAY_MIN(*NAME*, *SPLAY_HEAD *head*);

*struct TYPE **
SPLAY_MAX(*NAME*, *SPLAY_HEAD *head*);

*struct TYPE **
SPLAY_FIND(*NAME*, *SPLAY_HEAD *head*, *struct TYPE *elm*);

*struct TYPE **
SPLAY_LEFT(*struct TYPE *elm*, *SPLAY_ENTRY NAME*);

*struct TYPE **
SPLAY_RIGHT(*struct TYPE *elm*, *SPLAY_ENTRY NAME*);

SPLAY_FOREACH(*VARNAME*, *NAME*, *SPLAY_HEAD *head*);

void
SPLAY_INIT(*SPLAY_HEAD *head*);

*struct TYPE **
SPLAY_INSERT(*NAME*, *SPLAY_HEAD *head*, *struct TYPE *elm*);

*struct TYPE **
SPLAY_REMOVE(*NAME*, *SPLAY_HEAD *head*, *struct TYPE *elm*);

RB_PROTOTYPE(*NAME*, *TYPE*, *FIELD*, *CMP*);

RB_GENERATE(*NAME*, *TYPE*, *FIELD*, *CMP*);

RB_ENTRY(*TYPE*);

RB_HEAD(*HEADNAME*, *TYPE*);

RB_INITIALIZER(*RB_HEAD *head*);

*struct TYPE **
RB_ROOT(*RB_HEAD *head*);

bool
RB_EMPTY(*RB_HEAD *head*);

*struct TYPE **
RB_NEXT(*NAME*, *RB_HEAD *head*, *struct TYPE *elm*);

*struct TYPE **
RB_MIN(*NAME*, *RB_HEAD *head*);

*struct TYPE **
RB_MAX(*NAME*, *RB_HEAD *head*);

*struct TYPE **
RB_FIND(*NAME*, *RB_HEAD *head*, *struct TYPE *elm*);

*struct TYPE **
RB_LEFT(*struct TYPE *elm*, *RB_ENTRY NAME*);

*struct TYPE **
RB_RIGHT(*struct TYPE *elm*, *RB_ENTRY NAME*);

*struct TYPE **
RB_PARENT(*struct TYPE *elm*, *RB_ENTRY NAME*);

RB_FOREACH(*VARNAME*, *NAME*, *RB_HEAD *head*);

void
RB_INIT(*RB_HEAD *head*);

*struct TYPE **
RB_INSERT(*NAME*, *RB_HEAD *head*, *struct TYPE *elm*);

*struct TYPE **
RB_REMOVE(*NAME*, *RB_HEAD *head*, *struct TYPE *elm*);

DESCRIPTION

These macros define data structures for different types of trees: splay trees and red-black trees.

In the macro definitions, *TYPE* is the name tag of a user defined structure that must contain a field of type SPLAY_ENTRY, or RB_ENTRY, named *ENTRYNAME*. The argument *HEADNAME* is the name tag of a user defined structure that must be declared using the macros **SPLAY_HEAD**() or **RB_HEAD**(). The argument *NAME* has to be a unique name prefix for every tree that is defined.

The function prototypes are declared with either SPLAY_PROTOTYPE or RB_PROTOTYPE. The function bodies are generated with either SPLAY_GENERATE or RB_GENERATE. See the examples below for further explanation of how these macros are used.

SPLAY TREES

A splay tree is a self-organizing data structure. Every operation on the tree causes a splay to happen. The splay moves the requested node to the root of the tree and partly rebalances it.

This has the benefit that request locality causes faster lookups as the requested nodes move to the top of the tree. On the other hand, every lookup causes memory writes.

The Balance Theorem bounds the total access time for m operations and n inserts on an initially empty tree as O((m + n)lg n). The amortized cost for a sequence of m accesses to a splay tree is O(lg n).

A splay tree is headed by a structure defined by the **SPLAY_HEAD**() macro. A *SPLAY_HEAD* structure is declared as follows:

```
SPLAY_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and struct *TYPE* is the type of the elements to be inserted into the tree.

The **SPLAY_ENTRY**() macro declares a structure that allows elements to be connected in the tree.

In order to use the functions that manipulate the tree structure, their prototypes need to be declared with the **SPLAY_PROTOTYPE**() macro, where *NAME* is a unique identifier for this particular tree. The *TYPE* argument is the type of the structure that is being managed by the tree. The *FIELD* argument is the name of the element defined by **SPLAY_ENTRY**().

The function bodies are generated with the **SPLAY_GENERATE**() macro. It takes the same arguments as the **SPLAY_PROTOTYPE**() macro, but should be used only once.

Finally, the *CMP* argument is the name of a function used to compare trees noded with each other. The function takes two arguments of type *struct TYPE **. If the first argument is smaller than the second, the function returns a value smaller than zero. If they are equal, the function returns zero. Otherwise, it should return a value greater than zero. The compare function defines the order of the tree elements.

The **SPLAY_INIT**() macro initializes the tree referenced by *head*.

The splay tree can also be initialized statically by using the **SPLAY_INITIALIZER**() macro like this:

```
SPLAY_HEAD(HEADNAME, TYPE) head = SPLAY_INITIALIZER(&head);
```

The **SPLAY_INSERT**() macro inserts the new element *elm* into the tree.

The **SPLAY_REMOVE**() macro removes the element *elm* from the tree pointed by *head*.

The **SPLAY_FIND**() macro can be used to find a particular element in the tree.

```
struct TYPE find, *res;
find.key = 30;
res = SPLAY_FIND(NAME, head, &find);
```

The **SPLAY_ROOT**(), **SPLAY_MIN**(), **SPLAY_MAX**(), and **SPLAY_NEXT**() macros can be used to traverse the tree:

```
for (np = SPLAY_MIN(NAME, &head); np != NULL; np = SPLAY_NEXT(NAME, &head, np))
```

Or, for simplicity, one can use the **SPLAY_FOREACH**() macro:

```
SPLAY_FOREACH(np, NAME, head)
```

The **SPLAY_EMPTY**() macro should be used to check whether a splay tree is empty.

RED-BLACK TREES

A red-black tree is a binary search tree with the node color as an extra attribute. It fulfills a set of conditions:

1. every search path from the root to a leaf consists of the same number of black nodes,

2. each red node (except for the root) has a black parent,
3. each leaf node is black.

Every operation on a red-black tree is bounded as O(lg n). The maximum height of a red-black tree is 2lg (n+1).

A red-black tree is headed by a structure defined by the **RB_HEAD**() macro. A *RB_HEAD* structure is declared as follows:

```
RB_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and struct *TYPE* is the type of the elements to be inserted into the tree.

The **RB_ENTRY**() macro declares a structure that allows elements to be connected in the tree.

In order to use the functions that manipulate the tree structure, their prototypes need to be declared with the **RB_PROTOTYPE**() macro, where *NAME* is a unique identifier for this particular tree. The *TYPE* argument is the type of the structure that is being managed by the tree. The *FIELD* argument is the name of the element defined by **RB_ENTRY**().

The function bodies are generated with the **RB_GENERATE**() macro. It takes the same arguments as the **RB_PROTOTYPE**() macro, but should be used only once.

Finally, the *CMP* argument is the name of a function used to compare trees noded with each other. The function takes two arguments of type *struct TYPE* *. If the first argument is smaller than the second, the function returns a value smaller than zero. If they are equal, the function returns zero. Otherwise, it should return a value greater than zero. The compare function defines the order of the tree elements.

The **RB_INIT**() macro initializes the tree referenced by *head*.

The redblack tree can also be initialized statically by using the **RB_INITIALIZER**() macro like this:

```
RB_HEAD(HEADNAME, TYPE) head = RB_INITIALIZER(&head);
```

The **RB_INSERT**() macro inserts the new element *elm* into the tree.

The **RB_REMOVE**() macro removes the element *elm* from the tree pointed by *head*.

The **RB_FIND**() macro can be used to find a particular element in the tree.

```
struct TYPE find, *res;
find.key = 30;
res = RB_FIND(NAME, head, &find);
```

The **RB_ROOT**(), **RB_MIN**(), **RB_MAX**(), and **RB_NEXT**() macros can be used to traverse the tree:

```
for (np = RB_MIN(NAME, &head); np != NULL; np = RB_NEXT(NAME, &head, np))
```

Or, for simplicity, one can use the **RB_FOREACH**() macro:

```
RB_FOREACH(np, NAME, head)
```

The **RB_EMPTY**() macro should be used to check whether a red-black tree is empty.

NOTES

Trying to free a tree in the following way is a common error:

```
SPLAY_FOREACH(var, NAME, head) {
        SPLAY_REMOVE(NAME, head, var);
        free(var);
}
```

```
free(head);
```

Since *var* is free'd, the **FOREACH**() macro refers to a pointer that may have been reallocated already. Proper code needs a second variable.

```
for (var = SPLAY_MIN(NAME, head); var != NULL; var = nxt) {
        nxt = SPLAY_NEXT(NAME, head, var);
        SPLAY_REMOVE(NAME, head, var);
        free(var);
}
```

Both **RB_INSERT**() and **SPLAY_INSERT**() return NULL if the element was inserted in the tree successfully, otherwise they return a pointer to the element with the colliding key.

Accordingly, **RB_REMOVE**() and **SPLAY_REMOVE**() return the pointer to the removed element, otherwise they return NULL to indicate an error.

AUTHORS

The author of the tree macros is Niels Provos.

NAME

trunc, **truncf** — nearest integral value with magnitude less than or equal to |x|

LIBRARY

Math Library (libm, −lm)

SYNOPSIS

```
#include <math.h>

double
trunc(double x);

float
truncf(float x);
```

DESCRIPTION

The **trunc**() and **truncf**() functions return the nearest integral value with magnitude less than or equal to |*x*|. They are equivalent to **rint**() and **rintf**() respectively, in the FP_RZ rounding mode.

SEE ALSO

ceil(3), floor(3), fpsetround(3), math(3), nextafter(3), rint(3), round(3)

STANDARDS

The **trunc**() and **truncf**() functions conform to ISO/IEC 9899:1999 (“ISO C99”).

NAME

tsearch, tfind, tdelete, twalk — manipulate binary search trees

SYNOPSIS

```
#include <search.h>

void *
tdelete(const void * restrict key, void ** restrict rootp,
    int (*compar) (const void *, const void *));

void *
tfind(const void *key, const void * const *rootp,
    int (*compar) (const void *, const void *));

void *
tsearch(const void *key, void **rootp,
    int (*compar) (const void *, const void *));

void
twalk(const void *root, void (*action) (const void *, VISIT, int));
```

DESCRIPTION

The **tdelete**(), **tfind**(), **tsearch**(), and **twalk**() functions manage binary search trees based on algorithms T and D from Knuth (6.2.2). The comparison function passed in by the user has the same style of return values as strcmp(3).

tfind() searches for the datum matched by the argument *key* in the binary tree rooted at *rootp*, returning a pointer to the datum if it is found and NULL if it is not.

tsearch() is identical to **tfind**() except that if no match is found, *key* is inserted into the tree and a pointer to it is returned. If *rootp* points to a NULL value a new binary search tree is created.

tdelete() deletes a node from the specified binary search tree and returns a pointer to the parent of the node to be deleted. It takes the same arguments as **tfind**() and **tsearch**(). If the node to be deleted is the root of the binary search tree, *rootp* will be adjusted.

twalk() walks the binary search tree rooted in and calls the function *action* on each node. *Action* is called with three arguments: a pointer to the current node, a value from the enum **typedef enum { preorder, postorder, endorder, leaf } VISIT;** specifying the traversal type, and a node level (where level zero is the root of the tree).

RETURN VALUES

The **tsearch**() function returns NULL if allocation of a new node fails (usually due to a lack of free memory).

tfind(), **tsearch**(), and **tdelete**() return NULL if *rootp* is NULL or the datum cannot be found.

The **twalk**() function returns no value.

SEE ALSO

bsearch(3), hsearch(3), lsearch(3)

NAME

ttyaction — ttyaction utility function

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

int
ttyaction(char *ttyname, char *action, char *username);
```

DESCRIPTION

The **ttyaction**() function is used by `login`(1), `getty`(8), `telnetd`(8) and `rlogind`(8) to execute site-specific commands when a login session begins and ends.

The **ttyaction**() function scans the `/etc/ttyaction` file for any records that match the current *ttyname* and *action* parameters, and for each matching record, runs the shell command shown in that record. The record format is described in `ttyaction`(5). The parameter *username* is the name of the new owner of the *ttyname* device. Note that the *ttyname* parameter may be passed as a fully qualified pathname, and the **ttyaction**() function will skip the leading "/dev/" part of the string. (This is a convenience for login and getty.)

RETURN VALUES

ttyaction() returns the status of the last command it executed, or zero if no matching commands were found.

FILES

`/dev/*`
`/etc/ttyaction`

SEE ALSO

`ttyaction`(5)

AUTHORS

Gordon W. Ross ⟨gwr@NetBSD.org⟩,
Chris G. Demetriou ⟨cgd@NetBSD.org⟩,
Ty Sarna ⟨tsarna@endicor.com⟩.

BUGS

There should be some *other* mechanism to allow selection of different access control policies on a per-line basis. It has been suggested that the same **ttyaction** mechanism should also be used for determining access control, but it was decided (after much discussion) that **ttyaction** should only describe actions to be performed *after* the system has decided to change the ownership of some tty. Access control policies will be handled by a separate mechanism.

NAME

ttymsg — ttymsg utility function

LIBRARY

System Utilities Library (libutil, −lutil)

SYNOPSIS

```
#include <util.h>

char *
ttymsg(struct iovec *iov, int iovlen, const char *tty, int tmout);
```

DESCRIPTION

The **ttymsg**() function is used by programs such as talkd(8), syslogd(8), wall(1), etc., to display the contents of a uio structure on a terminal. **ttymsg**() forks and finishes in the child if the write would block after waiting up to *tmout* seconds.

RETURN VALUES

ttymsg() returns a pointer to an error string on unexpected error; the string is not newline-terminated. Various "normal" errors are ignored (exclusive-use, lack of permission, etc.).

SEE ALSO

writev(2)

NAME

ttyname, **ttyname_r**, **isatty**, **ttyslot** — get name of associated terminal (tty) from file descriptor

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

char *
ttyname(int fd);

int
ttyname_r(int fd, char *buf, size_t len);

int
isatty(int fd);

#include <stdlib.h>

int
ttyslot();
```

DESCRIPTION

These functions operate on the system file descriptors for terminal type devices. These descriptors are not related to the standard I/O `FILE` typedef, but refer to the special device files found in `/dev` and named `/dev/tty`*xx* and for which an entry exists in the initialization file `/etc/ttys` (see `ttys`(5)), or for pseudo-terminal devices created in ptyfs and named `/dev/pts/`*n*.

The **isatty**() function determines if the file descriptor *fd* refers to a valid terminal type device.

The **ttyname**() function gets the related device name of a file descriptor for which **isatty**() is true. The **ttyname_r**() is the reentrant version of the above, and it places the results in *buf*. If there is not enough space to place the results (indicated by *len*), then it returns −1 and *errno* is set to indicate the error.

The **ttyslot**() function fetches the current process' control terminal number from the `ttys`(5) file entry. If the terminal is a pseudo-terminal, and there is no special entry in the `ttys`(5) file for it, the slot number returned is 1 + (last slot number) + minor(tty). This will return a consistent and unique number for each pseudo-terminal device without requiring one to enumerate all of them in `ttys`(5).

IMPLEMENTATION NOTES

As an optimisation, these functions attempt to obtain information about all devices from the `/var/run/dev.db` database, if it exists. If the database exists but is out of date, then these functions may produce incorrect results. The database should be updated using the `dev_mkdb`(8) command.

RETURN VALUES

The **ttyname**() function returns the NUL-terminated name if the device is found and **isatty**() is true; otherwise a `NULL` pointer is returned and *errno* is set to indicate the error.

The **ttyname_r**() functions returns 0 on success and −1 on failure with *errno* set to indicate the error.

The **isatty**() function returns 1 if *fd* is associated with a terminal device; otherwise it returns 0 and *errno* is set to indicate the error.

The **ttyslot**() function returns the unit number of the device file if found; otherwise the value zero is returned.

FILES

`/dev/*`
`/etc/ttys`

ERRORS

The **ttyname**(), **ttyname_r**(), and **isatty**() functions will fail if:

[EBADF] The *fd* argument is not a valid file descriptor.

[ENOTTY] The *fd* argument does not refer to a terminal device.

The **ttyname_r**() function will also fail if:

[ERANGE] The buffer provided is not large enough to fit the result.

[ENOENT] The terminal device is not found. This can happen if the device node has been removed after it was opened.

SEE ALSO

ioctl(2), ttys(5), dev_mkdb(8)

STANDARDS

The **ttyname**() and **isatty**() functions conform to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

isatty(), **ttyname**(), and **ttyslot**() functions appeared in Version 7 AT&T UNIX.

BUGS

The **ttyname**() function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to **ttyname**() will modify the same object.

NAME

tzset — initialize time conversion information

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <time.h>

void
tzset(void);
```

DESCRIPTION

The **tzset**() function uses the value of the environment variable TZ to set time conversion information used by localtime(3). If TZ does not appear in the environment, the best available approximation to local wall clock time, as specified by the tzfile(5) format file /etc/localtime is used by localtime(3). If TZ appears in the environment but its value is a null string, Coordinated Universal Time (UTC) is used (without leap second correction). If TZ appears in the environment and its value is not a null string:

- if the value begins with a colon, it is used as a pathname of a file from which to read the time conversion information;
- if the value does not begin with a colon, it is first used as the pathname of a file from which to read the time conversion information, and, if that file cannot be read, is used directly as a specification of the time conversion information.

When TZ is used as a pathname, if it begins with a slash, it is used as an absolute pathname; otherwise, it is used as a pathname relative to /usr/share/zoneinfo. The file must be in the format specified in tzfile(5).

When TZ is used directly as a specification of the time conversion information, it must have the following syntax (spaces inserted for clarity):

```
stdoffset[dst[offset][,rule]]
```

where:

std and **dst** Three or more bytes that are the designation for the standard (**std**) or summer (**dst**) time zone. Only **std** is required; if **dst** is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except a leading colon (:), digits, comma (,), minus (-), plus (+), and ASCII NUL are allowed.

offset Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The **offset** has the form:

```
hh[:mm[:ss]]
```

The minutes (**mm**) and seconds (**ss**) are optional. The hour (**hh**) is required and may be a single digit. The **offset** following **std** is required. If no **offset** follows **dst**, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between zero and 24, and the minutes (and seconds) — if present — between zero and 59. If preceded by a "-" the time zone shall be east of the Prime Meridian; otherwise it shall be west (which may be indicated by an optional preceding "+").

rule Indicates when to change to and back from summer time. The **rule** has the form:

```
date/time,date/time
```

where the first **date** describes when the change from standard to summer time occurs and the second **date** describes when the change back happens. Each **time** field describes when, in current local time, the change to the other time is made. The format of *date* is one of the following:

J*n* The Julian day *n* ($1 \le n \le 365$). Leap days are not counted; that is, in all years — including leap years — February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

n The zero-based Julian day ($0 \le n \le 365$). Leap days are counted, and it is possible to refer to February 29.

M*m.n.d* The *d*'th day ($0 \le d \le 6$) of week *n* of month *m* of the year ($1 \le n \le 5$, $1 \le m \le 12$, where week 5 means "the last *d* day in month *m*" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*'th day occurs. Day zero is Sunday.

The **time** has the same format as **offset** except that no leading sign "-" or "+" is allowed. The default, if **time** is not given, is **02:00:00**.

If no **rule** is present in TZ, the rules specified by the tzfile(5) format file posixrules in /usr/share/zoneinfo are used, with the standard and summer time offsets from UTC replaced by those specified by the **offset** values in TZ.

For compatibility with System V Release 3.1, a semicolon (;) may be used to separate the **rule** from the rest of the specification.

If the TZ environment variable does not specify a tzfile(5) format file and cannot be interpreted as a direct specification, UTC is used.

FILES

/etc/localtime	local time zone file
/usr/share/zoneinfo	time zone information directory
/usr/share/zoneinfo/posixrules	used with POSIX-style TZ's
/usr/share/zoneinfo/GMT	for UTC leap seconds

If /usr/share/zoneinfo/GMT is absent, UTC leap seconds are loaded from /usr/share/zoneinfo/posixrules.

SEE ALSO

ctime(3), getenv(3), strftime(3), time(3), tzfile(5)

STANDARDS

The tzset() function conforms to IEEE Std 1003.1-1988 ("POSIX.1").

NAME

ualarm — schedule signal after specified time

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

useconds_t
ualarm(useconds_t microseconds, useconds_t interval);
```

DESCRIPTION

This is a simplified interface to setitimer(2).

The **ualarm**() function waits a count of *microseconds* before asserting the terminating signal SIGALRM. System activity or time used in processing the call may cause a slight delay.

If the *interval* argument is non-zero, the SIGALRM signal will be sent to the process every *interval* microseconds after the timer expires (e.g. after *microseconds* microseconds have passed).

RETURN VALUES

When the signal has successfully been caught, **ualarm**() returns the amount of time left on the clock. The maximum number of *microseconds* allowed is 2147483647. If there is an error setting the timer, **ualarm**() returns ((useconds_t) -1).

SEE ALSO

getitimer(2), setitimer(2), sigaction(2), sigsuspend(2), alarm(3), signal(3), sigvec(3), sleep(3), usleep(3)

STANDARDS

The **ualarm**() functions conforms to X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”).

HISTORY

The **ualarm**() function appeared in 4.3BSD.

NAME

ulimit — get and set process limits

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <ulimit.h>

long int
ulimit(int cmd, ...);
```

DESCRIPTION

The **ulimit**() function provides a method to query or alter resource limits of the calling process. The method to be performed is specified by the *cmd* argument; possible values are:

UL_GETFSIZE — Return the soft file size limit of the process. The value returned is in units of 512-byte blocks. If the result cannot be represented in an object of type *long int*, the result is unspecified.

UL_SETFSIZE — Set the hard and soft file size limits of the process to the value of the second argument passed, which is in units of 512-byte blocks, and which is expected to be of type *long int*. The new file size limit of the process is returned. Any process may decrease the limit, but raising it is only permitted if the caller is the super-user.

If successful, the **ulimit**() function will not change the setting of *errno*.

The **ulimit**() function is an obsolete interface; applications are encouraged to use getrlimit(2) and setrlimit(2) instead.

RETURN VALUES

If successful, the **ulimit**() function returns the value of the requested limit. Otherwise, it returns −1, sets *errno* to indicate an error, and the limit is not changed. Therefore, to detect an error condition applications should set *errno* to 0, call **ulimit**(), and check if −1 is returned and *errno* is non-zero.

ERRORS

The **ulimit**() function will fail if:

[EINVAL] — The *cmd* argument is not valid.

[EPERM] — It was attempted to increase a limit, and the caller is not the super-user.

SEE ALSO

getrlimit(2), setrlimit(2)

STANDARDS

The **ulimit**() function conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

NAME

uname — get system identification

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/utsname.h>

int
uname(struct utsname *name);
```

DESCRIPTION

The **uname**() function stores nul-terminated strings of information identifying the current system into the structure referenced by *name*.

The utsname structure is defined in the <sys/utsname.h> header file, and contains the following members:

sysname	Name of the operating system implementation.
nodename	Network name of this machine.
release	Release level of the operating system.
version	Version level of the operating system.
machine	Machine hardware platform.

RETURN VALUES

If **uname** is successful, 0 is returned, otherwise, −1 is returned and *errno* is set appropriately.

ERRORS

The **uname**() function may fail and set *errno* for any of the errors specified for the library functions sysctl(3).

SEE ALSO

uname(1), sysctl(3)

STANDARDS

The **uname**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

The **uname** function first appeared in 4.4BSD.

NAME

ungetc — un-get character from input stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>

int
ungetc(int c, FILE *stream);
```

DESCRIPTION

The **ungetc**() function pushes the character *c* (converted to an unsigned char) back onto the input stream pointed to by *stream*. The pushed-backed characters will be returned by subsequent reads on the stream (in reverse order). A successful intervening call, using the same stream, to one of the file positioning functions (`fseek`(3), `fsetpos`(3), or `rewind`(3)) will discard the pushed back characters.

One character of push-back is guaranteed, but as long as there is sufficient memory, an effectively infinite amount of pushback is allowed.

If a character is successfully pushed-back, the end-of-file indicator for the stream is cleared.

RETURN VALUES

The **ungetc**() function returns the character pushed-back after the conversion, or `EOF` if the operation fails. If the value of the argument *c* character equals `EOF`, the operation will fail and the stream will remain unchanged.

SEE ALSO

`fseek`(3), `getc`(3), `setvbuf`(3)

STANDARDS

The **ungetc**() function conforms to ANSI X3.159-1989 (“ANSI C89”).

NAME

ungetwc — un-get wide-character from input stream

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t
ungetwc(wint_t wc, FILE *stream);
```

DESCRIPTION

The **ungetwc**() function pushes the wide-character *wc* (converted to an wchar_t) back onto the input stream pointed to by *stream*. The pushed-backed wide-characters will be returned by subsequent reads on the stream (in reverse order). A successful intervening call, using the same stream, to one of the file positioning functions fseek(3), fsetpos(3), or rewind(3) will discard the pushed back wide-characters.

One wide-character of push-back is guaranteed, but as long as there is sufficient memory, an effectively infinite amount of pushback is allowed.

If a character is successfully pushed-back, the end-of-file indicator for the stream is cleared.

RETURN VALUES

The **ungetwc**() function returns the wide-character pushed-back after the conversion, or WEOF if the operation fails. If the value of the argument *c* character equals WEOF, the operation will fail and the stream will remain unchanged.

SEE ALSO

fseek(3), getwc(3)

STANDARDS

The **ungetwc**() function conforms to ISO/IEC 9899:1999 (“ISO C99”).

BUGS

The current implementation uses a fixed sized ungetwc-buffer.

NAME

unlockpt — unlock the slave pseudo-terminal device

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
unlockpt(int fildes);
```

DESCRIPTION

The **unlockpt**() unlocks access to the pseudo-terminal device corresponding to the master pseudo-terminal device associated with *fildes*. Conforming applications must call this function before opening the slave pseudo-terminal device.

RETURN VALUES

If successful, **unlockpt**() returns 0; otherwise a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

The **unlockpt**() function will fail if:

[EACCESS] the corresponding pseudo-terminal device could not be accessed.

[EBADF] *fildes* is not a valid descriptor.

[EINVAL] *fildes* is not associated with a master pseudo-terminal device.

NOTES

In NetBSD **unlockpt**() does nothing.

SEE ALSO

ioctl(2), grantpt(3), posix_openpt(3), ptsname(3)

STANDARDS

The **unlockpt**() function conforms to IEEE Std 1003.1-2001 (“POSIX.1”). Its first release was in X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”).

NAME

unvis, **strunvis** — decode a visual representation of characters

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <vis.h>

int
unvis(char *cp, int c, int *astate, int flag);

int
strunvis(char *dst, const char *src);

int
strunvisx(char *dst, const char *src, int flag);
```

DESCRIPTION

The **unvis**(), **strunvis**() and **strunvisx**() functions are used to decode a visual representation of characters, as produced by the vis(3) function, back into the original form.

The **unvis**() function is called with successive characters in *c* until a valid sequence is recognized, at which time the decoded character is available at the character pointed to by *cp*.

The **strunvis**() function decodes the characters pointed to by *src* into the buffer pointed to by *dst*. The **strunvis**() function simply copies *src* to *dst*, decoding any escape sequences along the way, and returns the number of characters placed into *dst*, or −1 if an invalid escape sequence was detected. The size of *dst* should be equal to the size of *src* (that is, no expansion takes place during decoding).

The **strunvisx**() function does the same as the **strunvis**() function, but it allows you to add a flag that specifies the style the string *src* is encoded with. Currently, the only supported flag is VIS_HTTPSTYLE.

The **unvis**() function implements a state machine that can be used to decode an arbitrary stream of bytes. All state associated with the bytes being decoded is stored outside the **unvis**() function (that is, a pointer to the state is passed in), so calls decoding different streams can be freely intermixed. To start decoding a stream of bytes, first initialize an integer to zero. Call **unvis**() with each successive byte, along with a pointer to this integer, and a pointer to a destination character. The **unvis**() function has several return codes that must be handled properly. They are:

0 (zero)	Another character is necessary; nothing has been recognized yet.
UNVIS_VALID	A valid character has been recognized and is available at the location pointed to by cp.
UNVIS_VALIDPUSH	A valid character has been recognized and is available at the location pointed to by cp; however, the character currently passed in should be passed in again.
UNVIS_NOCHAR	A valid sequence was detected, but no character was produced. This return code is necessary to indicate a logical break between characters.
UNVIS_SYNBAD	An invalid escape sequence was detected, or the decoder is in an unknown state. The decoder is placed into the starting state.

When all bytes in the stream have been processed, call **unvis**() one more time with flag set to UNVIS_END to extract any remaining character (the character passed in is ignored).

The *flag* argument is also used to specify the encoding style of the source. If set to VIS_HTTPSTYLE, **unvis**() will decode URI strings as specified in RFC 1808.

The following code fragment illustrates a proper use of **unvis**().

```
int state = 0;
char out;

while ((ch = getchar()) != EOF) {
again:
        switch(unvis(&out, ch, &state, 0)) {
        case 0:
        case UNVIS_NOCHAR:
                break;
        case UNVIS_VALID:
                (void)putchar(out);
                break;
        case UNVIS_VALIDPUSH:
                (void)putchar(out);
                goto again;
        case UNVIS_SYNBAD:
                errx(EXIT_FAILURE, "Bad character sequence!");
        }
}
if (unvis(&out, '\0', &state, UNVIS_END) == UNVIS_VALID)
        (void)putchar(out);
```

SEE ALSO

unvis(1), vis(1), vis(3)

R. Fielding, *Relative Uniform Resource Locators*, RFC1808.

HISTORY

The **unvis**() function first appeared in 4.4BSD.

NAME

usbhid, **hid_get_report_desc**, **hid_use_report_desc**, **hid_dispose_report_desc**, **hid_start_parse**, **hid_end_parse**, **hid_get_item**, **hid_report_size**, **hid_locate**, **hid_usage_page**, **hid_usage_in_page**, **hid_init**, **hid_get_data**, **hid_set_data** — USB HID access routines

LIBRARY

USB Human Interface Devices Library (libusbhid, −lusbhid)

SYNOPSIS

```
#include <usbhid.h>

report_desc_t
hid_get_report_desc(int file);

report_desc_t
hid_use_report_desc(const uint8_t *data, unsigned int size);

void
hid_dispose_report_desc(report_desc_t d);

hid_data_t
hid_start_parse(report_desc_t d, int kindset, int id);

void
hid_end_parse(hid_data_t s);

int
hid_get_item(hid_data_t s, hid_item_t *h);

int
hid_report_size(report_desc_t d, hid_kind_t k, int id);

int
hid_locate(report_desc_t d, u_int usage, hid_kind_t k, hid_item_t *h,
    int id);

char *
hid_usage_page(int i);

char *
hid_usage_in_page(u_int u);

int
hid_parse_usage_page(const char *);

char *
hid_parse_usage_in_page(const char *);

void
hid_init(char *file);

int
hid_get_data(void *data, hid_item_t *h);

void
hid_set_data(void *data, hid_item_t *h, u_int data);
```

DESCRIPTION

The **usbhid** library provides routines to extract data from USB Human Interface Devices.

INTRODUCTION

USB HID devices send and receive data laid out in a device dependent way. The **usbhid** library contains routines to extract the *report descriptor* which contains the data layout information and then use this information.

The routines can be divided into four parts: extraction of the descriptor, parsing of the descriptor, translating to/from symbolic names, and data manipulation.

DESCRIPTOR FUNCTIONS

A report descriptor can be obtained by calling **hid_get_report_desc**() with a file descriptor obtained by opening a uhid(4) device. Alternatively a data buffer containing the report descriptor can be passed into **hid_use_report_desc**(). The data is copied into an internal structure. When the report descriptor is no longer needed it should be freed by calling **hid_dispose_report_desc**(). The type *report_desc_t* is opaque and should be used when calling the parsing functions. If **hid_dispose_report_desc**() fails it will return *NULL*.

DESCRIPTOR PARSING FUNCTIONS

To parse the report descriptor the **hid_start_parse**() function should be called with a report descriptor and a set that describes which items that are interesting. The set is obtained by or-ing together values *(1 << k)* where *k* is an item of type *hid_kind_t*. The report ID (if present) is given by *id*. The function returns *NULL* if the initialization fails, otherwise an opaque value to be used in subsequent calls. After parsing the **hid_end_parse**() function should be called to free internal data structures.

To iterate through all the items in the report descriptor, **hid_get_item**() should be called while it returns a value greater than 0. When the report descriptor ends it will return 0; a syntax error within the report descriptor will cause a return value less than 0. The struct pointed to by *h* will be filled with the relevant data for the item. The definition of *hid_item_t* can be found in ⟨usbhid.h⟩ and the meaning of the components in the USB HID documentation.

Data should be read/written to the device in the size of the report. The size of a report (of a certain kind) can be computed by the **hid_report_size**() function. If the report is prefixed by an ID byte it is given by *id*.

To locate a single item the **hid_locate**() function can be used. It should be given the usage code of the item and its kind and it will fill the item and return non-zero if the item was found.

NAME TRANSLATION FUNCTIONS

The function **hid_usage_page**() will return the symbolic name of a usage page, and the function **hid_usage_in_page**() will return the symbolic name of the usage within the page. Both these functions may return a pointer to static data.

The functions **hid_parse_usage_page**() and **hid_parse_usage_in_page**() are the inverses of **hid_usage_page**() and **hid_usage_in_page**(). They take a usage string and return the number of the usage, or -1 if it cannot be found.

Before any of these functions can be called the usage table must be parsed, this is done by calling **hid_init**() with the name of the table. Passing *NULL* to this function will cause it to use the default table.

DATA EXTRACTION FUNCTIONS

Given the data obtained from a HID device and an item in the report descriptor the **hid_get_data**() function extracts the value of the item. Conversely **hid_set_data**() can be used to put data into a report (which must be zeroed first).

FILES

`/usr/share/misc/usb_hid_usages` The default HID usage table.

SEE ALSO

`uhid`(4), `usb`(4)

The USB specifications can be found at `http://www.usb.org/developers/docs.html`.

HISTORY

The **usbhid** library first appeared in NetBSD 1.5 as **usb** library and was renamed to **usbhid** for NetBSD 1.6.

BUGS

This man page is woefully incomplete.

NAME

usleep — suspend execution for interval of microseconds

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <unistd.h>

int
usleep(useconds_t microseconds);
```

DESCRIPTION

The **usleep**() function suspends execution of the calling process until either the number of microseconds specified by *microseconds* have elapsed or a signal is delivered to the calling process and its action is to invoke a signal catching function or to terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system.

The *microseconds* argument must be less than 1,000,000. If the value of *microseconds* is 0, then the call has no effect.

RETURN VALUES

On successful completion, **usleep**() returns 0. Otherwise, it returns −1 and sets *errno* to indicate the error.

ERRORS

The **usleep**() function may fail if:

[EINVAL] The *microseconds* interval specified 1,000,000 or more microseconds.

SEE ALSO

nanosleep(2), sleep(3)

STANDARDS

The **usleep**() function conforms to X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”).

HISTORY

The **usleep**() function appeared in 4.3BSD.

NAME

util — system utilities library

LIBRARY

System Utilities Library (libutil, −lutil)

DESCRIPTION

The **util** library is the system utilities library and contains various system-dependent utility routines used in a wide variety of system daemons. The abstracted functions are mostly related to pseudo-terminals and login accounting. These routines are NetBSD-specific and are not portable. Their use should be restricted. Declarations for these functions may be obtained from the include file `<util.h>`.

LIST OF FUNCTIONS

Name/Page	*Description*
disklabel_dkcksum.3	compute the checksum for a disklabel
disklabel_scan.3	scan a buffer for a valid disklabel
forkpty.3	tty utility function
getbootfile.3	get the name of the booted kernel file
getlabeloffset.3	get the sector number and offset of the disklabel
getlabelsector.3	get the sector number and offset of the disklabel
getmaxpartitions.3	get the maximum number of partitions allowed per disk
getrawpartition.3	get the system ‘‘raw’’ partition
login.3	login utility function
login_cap.3	query login.conf database about a user class
login_close.3	query login.conf database about a user class
login_getcapbool.3	query login.conf database about a user class
login_getcapnum.3	query login.conf database about a user class
login_getcapsize.3	query login.conf database about a user class
login_getcapstr.3	query login.conf database about a user class
login_getcaptime.3	query login.conf database about a user class
login_getclass.3	query login.conf database about a user class
login_tty.3	tty utility function
loginx.3	login utility function
logout.3	login utility function
logoutx.3	login utility function
logwtmp.3	login utility function
logwtmpx.3	login utility function
opendisk.3	open a disk partition
openpty.3	tty utility function
pidfile.3	write a daemon pid file
pidlock.3	locks based on files containing PIDs
pw_abort.3	passwd file update function
pw_copy.3	utility function for interactive passwd file updates
pw_edit.3	utility function for interactive passwd file updates
pw_error.3	utility function for interactive passwd file updates
pw_getconf.3	password encryption configuration access function
pw_getprefix.3	passwd file update function
pw_init.3	utility function for interactive passwd file updates
pw_lock.3	passwd file update function
pw_mkdb.3	passwd file update function
pw_prompt.3	utility function for interactive passwd file updates
pw_scan.3	utility function for interactive passwd file updates

pw_setprefix.3	passwd file update function
secure_path.3	determine if a file appears to be ‘‘secure’’
setclasscontext.3	query login.conf database about a user class
setusercontext.3	query login.conf database about a user class
snprintb.3	bitmask output conversion
sockaddr_snprintf.3	socket address formatting function
ttyaction.3	ttyaction utility function
ttylock.3	locks based on files containing PIDs
ttymsg.3	ttymsg utility function
ttyunlock.3	locks based on files containing PIDs

FILES

`/usr/lib/libutil.a`	static util library
`/usr/lib/libutil.so`	dynamic util library
`/usr/lib/libutil_p.a`	static util library compiled for profiling

NAME

utime — set file times

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <utime.h>

int
utime(const char *file, const struct utimbuf *timep);
```

DESCRIPTION

This interface is obsoleted by utimes(2)**.**

The **utime**() function sets the access and modification times of the named file.

If *timep* is NULL, the access and modification times are set to the current time. The calling process must be the owner of the file or have permission to write the file.

If *timep* is non-NULL, *time* is assumed to be a pointer to a utimbuf structure, as defined in ⟨utime.h⟩:

```
struct utimbuf {
        time_t actime;          /* Access time */
        time_t modtime;         /* Modification time */
};
```

The access time is set to the value of the actime member, and the modification time is set to the value of the modtime member. The times are measured in seconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970 Coordinated Universal Time (UTC). The calling process must be the owner of the file or be the super-user.

In either case, the inode-change-time of the file is set to the current time.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

utime() will fail if:

[EACCES]	Search permission is denied for a component of the path prefix; or the *times* argument is NULL and the effective user ID of the process does not match the owner of the file, and is not the super-user, and write access is denied.
[EFAULT]	*file* or *times* points outside the process's allocated address space.
[EINVAL]	The pathname contains a character with the high-order bit set.
[EIO]	An I/O error occurred while reading or writing the affected inode.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The *times* argument is not NULL and the calling process's effective user ID does not match the owner of the file and is not the super-user.

[EROFS] The file system containing the file is mounted read-only.

SEE ALSO

stat(2), utimes(2)

STANDARDS

The **utime**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

HISTORY

A **utime**() function appeared in Version 7 AT&T UNIX.

NAME

uuid_compare, **uuid_create**, **uuid_create_nil**, **uuid_equal**, **uuid_from_string**, **uuid_hash**, **uuid_is_nil**, **uuid_to_string**, **uuid_enc_le**, **uuid_dec_le**, **uuid_enc_be**, **uuid_dec_be** — Universally Unique Identifier routines

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <uuid.h>

int32_t
uuid_compare(const uuid_t *uuid1, const uuid_t *uuid2, uint32_t *status);

void
uuid_create(uuid_t *uuid, uint32_t *status);

void
uuid_create_nil(uuid_t *uuid, uint32_t *status);

int32_t
uuid_equal(const uuid_t *uuid1, const uuid_t *uuid2, uint32_t *status);

void
uuid_from_string(const char *str, uuid_t *uuid, uint32_t *status);

uint16_t
uuid_hash(const uuid_t *uuid, uint32_t *status);

int32_t
uuid_is_nil(const uuid_t *uuid, uint32_t *status);

void
uuid_to_string(const uuid_t *uuid, char **str, uint32_t *status);

void
uuid_enc_le(void *buf, const uuid_t *uuid);

void
uuid_dec_le(const void *buf, uuid_t *);

void
uuid_enc_be(void *buf, const uuid_t *uuid);

void
uuid_dec_be(const void *buf, uuid_t *);
```

DESCRIPTION

These routines provide for the creation and manipulation of Universally Unique Identifiers (UUIDs), also referred to as Globally Unique Identifiers (GUIDs).

The **uuid_compare**() function compares two UUIDs. It returns −1 if *uuid1* precedes *uuid2*, 0 if they are equal, or 1 if *uuid1* follows *uuid2*.

The **uuid_create**() function creates a new UUID. Storage for the new UUID must be pre-allocated by the caller.

The **uuid_create_nil**() function creates a nil-valued UUID. Storage for the new UUID must be pre-allocated by the caller.

The **uuid_equal**() function compares two UUIDs to determine if they are equal. It returns 1 if they are equal, and 0 if they are not equal.

The **uuid_from_string**() function parses a 36-character string representation of a UUID and converts it to binary representation. Storage for the UUID must be pre-allocated by the caller.

The **uuid_hash**() function generates a hash value for the specified UUID. Note that the hash value is not a cryptographic hash, and should not be assumed to be unique given two different UUIDs.

The **uuid_is_nil**() function returns 1 if the UUID is nil-valued and 0 if it is not.

The **uuid_to_string**() function converts a UUID from binary representation to string representation. Storage for the string is dynamically allocated and returned via the *str* argument. This pointer should be passed to free(3) to release the allocated storage when it is no longer needed.

The **uuid_enc_le**() and **uuid_enc_be**() functions encode a binary representation of a UUID into an octet stream in little-endian and big-endian byte-order, respectively. The destination buffer must be pre-allocated by the caller, and must be large enough to hold the 16-octet binary UUID.

The **uuid_dec_le**() and **uuid_dec_be**() functions decode a UUID from an octet stream in little-endian and big-endian byte-order, respectively.

RETURN VALUES

The **uuid_compare**(), **uuid_create**(), **uuid_create_nil**(), **uuid_equal**(), **uuid_from_string**(), **uuid_hash**(), **uuid_is_nil**(), and **uuid_to_string**() functions return successful or unsuccessful completion status in the *status* argument. Possible values are:

uuid_s_ok	The function completed successfully.
uuid_s_bad_version	The UUID does not have a known version.
uuid_s_invalid_string_uuid	The string representation of a UUID is not valid.
uuid_s_no_memory	Memory could not be allocated for the operation.

SEE ALSO

uuidgen(1), uuidgen(2)

STANDARDS

The **uuid_compare**(), **uuid_create**(), **uuid_create_nil**(), **uuid_equal**(), **uuid_from_string**(), **uuid_hash**(), **uuid_is_nil**(), and **uuid_to_string**() functions are compatible with the DCE 1.1 RPC specification.

HISTORY

The UUID functions first appeared in NetBSD 3.0.

NAME

valloc — aligned memory allocation function

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

void *
valloc(size_t size);
```

DESCRIPTION

Valloc is obsoleted by the current version of malloc(**3**)**, which aligns page-sized and larger allocations.**

The **valloc**() function allocates *size* bytes aligned on a page boundary. It is implemented by calling malloc(3) with a slightly larger request, saving the true beginning of the block allocated, and returning a properly aligned pointer.

RETURN VALUES

The **valloc**() function returns a pointer to the allocated space if successful; otherwise a null pointer is returned

HISTORY

The **valloc**() function appeared in 3.0BSD.

BUGS

A *vfree* function has not been implemented.

NAME

varargs — variable argument lists

SYNOPSIS

```
#include <varargs.h>

void
va_start(va_list ap);

type
va_arg(va_list ap, type);

void
va_end(va_list ap);
```

DESCRIPTION

These historic interfaces are provided to support compilation of existing programs only. New code should use the `stdarg`**(3) interfaces.**

A function may be called with a varying number of arguments of varying types. The include file ⟨`varargs.h`⟩ declares a type (*va_list*) and defines three macros for stepping through a list of arguments whose number and types are not known to the called function.

The called function must name an argument *va_alist*, which marks the start of the variable argument list, and which is naturally the last argument named. It is declared by *va_dcl*, which should not be followed by a semicolon. The called function also must declare an object of type *va_list* which is used by the macros **va_start**(), **va_arg**(), and **va_end**().

The **va_start**() macro initializes *ap* for subsequent use by **va_arg**() and **va_end**(), and must be called first.

It is possible for *va_alist* to be the only parameter to a function, resulting in it being possible for a function to have no fixed arguments preceding the variable argument list.

The **va_start**() macro returns no value.

The **va_arg**() macro expands to an expression that has the type and value of the next argument in the call. The parameter *ap* is the *va_list* *ap* initialized by **va_start**(). Each call to **va_arg**() modifies *ap* so that the next call returns the next argument. The parameter *type* is a type name specified so that the type of a pointer to an object that has the specified type can be obtained simply by adding a ∗ to *type*.

If there is no next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), random errors will occur.

The first use of the **va_arg**() macro after that of the **va_start**() macro returns the argument after *last*. Successive invocations return the values of the remaining arguments.

The **va_end**() macro handles a normal return from the function whose variable argument list was initialized by **va_start**().

The **va_end**() macro returns no value.

EXAMPLES

The function *foo* takes a string of format characters and prints out the argument associated with each format character based on the type.

```
void foo(fmt, va_alist)
        char *fmt;
        va_dcl
```

```
{
        va_list ap;
        int d;
        char c, *p, *s;

        va_start(ap);
        while (*fmt) {
                switch (*fmt++) {
                case 's':                       /* string */
                        s = va_arg(ap, char *);
                        printf("string %s\n", s);
                        break;
                case 'd':                       /* int */
                        d = va_arg(ap, int);
                        printf("int %d\n", d);
                        break;
                case 'c':                       /* char */
                        c = va_arg(ap, char);
                        printf("char %c\n", c);
                        break;
                }
        }
        va_end(ap);
}
```

SEE ALSO

`stdarg`(3)

STANDARDS

These historic macros were replaced in ANSI X3.159-1989 (“ANSI C89”) by the include file ⟨`stdarg.h`⟩; see `stdarg`(3) for its description.

COMPATIBILITY

These macros are *not* compatible with the new macros they were replaced by. In particular, it is not possible for a *stdarg* function to have no fixed arguments.

NAME

virtdir — Utility routines for virtual directories for refuse operations

SYNOPSIS

```
#include <virtdir.h>

int
virtdir_init(virtdir_t *tree, struct stat *dir, struct stat *file,
    struct stat *symlink);

int
virtdir_add(virtdir_t *tree, const char *name, size_t namesize,
    uint8_t type, char *target);

int
virtdir_del(virtdir_t *tree, const char *name, size_t namesize);

int
virtdir_find(virtdir_t *tree, const char *name, size_t namesize);

int
virtdir_find_tgt(virtdir_t *tree, const char *name, size_t namesize);

void
virtdir_drop(virtdir_t *tree);

VIRTDIR *
openvirtdir(virtdir_t *tree, const char *directory);

virt_dirent_t *
readvirtdir(VIRTDIR *dirp);

void
closevirtdir(VIRTDIR *dirp);
```

DESCRIPTION

virtdir provides virtual directory functionality for the benefit of refuse(3) file systems (and also for FUSE-based file systems).

It uses the framework provided by the puffs(3) subsystem, and, through that, the kernel interface provided by puffs(4).

The **virtdir** routines build up and manage a list of virtual directory entries. Each virtual directory entry is indexed by its full pathname within the file system. This is consistent with the way that refuse(3) locates directory entries - by full pathname.

The list of paths is sorted alphabetically. Each of these virtual directory entries has a distinct type - file ('f'), directory ('d'), or symbolic link ('l'). Additionally, an entry can point to a target - this is useful when modeling virtual directory entries which are symbolic links. The list contains three basic stat(2) structures, which contain basic information for file, directory and symbolic link entries. This information can be specified at initialization time, and customized within the individual getattr operation routines as specified by the individual file systems. The **virtdir** functionality can also make virtual directory entries available on a per-directory basis to the caller by means of routines analogous to opendir(3), readdir(3), and closedir(3). These are **openvirtdir**(), **readvirtdir**(), and **closevirtdir**(), respectively.

SEE ALSO

`puffs`(3), `refuse`(3), `puffs`(4)

HISTORY

An unsupported experimental version of **`virtdir`** first appeared in NetBSD 5.0.

AUTHORS

Alistair Crooks ⟨agc@NetBSD.org⟩

NAME

vis, **strvis**, **strvisx**, **svis**, **strsvis**, **strsvisx** — visually encode characters

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <vis.h>

char *
vis(char *dst, int c, int flag, int nextc);

int
strvis(char *dst, const char *src, int flag);

int
strvisx(char *dst, const char *src, size_t len, int flag);

char *
svis(char *dst, int c, int flag, int nextc, const char *extra);

int
strsvis(char *dst, const char *src, int flag, const char *extra);

int
strsvisx(char *dst, const char *src, size_t len, int flag,
    const char *extra);
```

DESCRIPTION

The **vis**() function copies into *dst* a string which represents the character *c*. If *c* needs no encoding, it is copied in unaltered. The string is null terminated, and a pointer to the end of the string is returned. The maximum length of any encoding is four characters (not including the trailing NUL); thus, when encoding a set of characters into a buffer, the size of the buffer should be four times the number of characters encoded, plus one for the trailing NUL. The flag parameter is used for altering the default range of characters considered for encoding and for altering the visual representation. The additional character, *nextc*, is only used when selecting the VIS_CSTYLE encoding format (explained below).

The **strvis**() and **strvisx**() functions copy into *dst* a visual representation of the string *src*. The **strvis**() function encodes characters from *src* up to the first NUL. The **strvisx**() function encodes exactly *len* characters from *src* (this is useful for encoding a block of data that may contain NUL's). Both forms NUL terminate *dst*. The size of *dst* must be four times the number of characters encoded from *src* (plus one for the NUL). Both forms return the number of characters in dst (not including the trailing NUL).

The functions **svis**(), **strsvis**(), and **strsvisx**() correspond to **vis**(), **strvis**(), and **strvisx**() but have an additional argument *extra*, pointing to a NUL terminated list of characters. These characters will be copied encoded or backslash-escaped into *dst*. These functions are useful e.g. to remove the special meaning of certain characters to shells.

The encoding is a unique, invertible representation composed entirely of graphic characters; it can be decoded back into the original form using the unvis(3) or strunvis(3) functions.

There are two parameters that can be controlled: the range of characters that are encoded (applies only to **vis**(), **strvis**(), and **strvisx**()), and the type of representation used. By default, all non-graphic characters, except space, tab, and newline are encoded. (See isgraph(3).) The following flags alter this:

VIS_SP Also encode space.

VIS_TAB Also encode tab.

VIS_NL Also encode newline.

VIS_WHITE Synonym for VIS_SP | VIS_TAB | VIS_NL.

VIS_SAFE Only encode "unsafe" characters. Unsafe means control characters which may cause common terminals to perform unexpected functions. Currently this form allows space, tab, newline, backspace, bell, and return - in addition to all graphic characters - unencoded.

(The above flags have no effect for **svis**(), **strsvis**(), and **strsvisx**(). When using these functions, place all graphic characters to be encoded in an array pointed to by *extra*. In general, the backslash character should be included in this array, see the warning on the use of the VIS_NOSLASH flag below).

There are four forms of encoding. All forms use the backslash character '\' to introduce a special sequence; two backslashes are used to represent a real backslash, except VIS_HTTPSTYLE that uses '%'. These are the visual formats:

(default) Use an 'M' to represent meta characters (characters with the 8th bit set), and use caret '^' to represent control characters see (iscntrl(3)). The following formats are used:

\^C Represents the control character 'C'. Spans characters \000 through \037, and \177 (as \^?).

\M-C Represents character 'C' with the 8th bit set. Spans characters \241 through \376.

\M^C Represents control character 'C' with the 8th bit set. Spans characters \200 through \237, and \377 (as \M^?).

\040 Represents ASCII space.

\240 Represents Meta-space.

VIS_CSTYLE Use C-style backslash sequences to represent standard non-printable characters. The following sequences are used to represent the indicated characters:

\a - BEL (007)
\b - BS (010)
\f - NP (014)
\n - NL (012)
\r - CR (015)
\s - SP (040)
\t - HT (011)
\v - VT (013)
\0 - NUL (000)

When using this format, the nextc parameter is looked at to determine if a NUL character can be encoded as '\0' instead of \000. If *nextc* is an octal digit, the latter representation is used to avoid ambiguity.

VIS_OCTAL Use a three digit octal sequence. The form is \ddd where *d* represents an octal digit.

VIS_HTTPSTYLE
Use URI encoding as described in RFC 1738. The form is %xx where *x* represents a hexadecimal digit.

There is one additional flag, `VIS_NOSLASH`, which inhibits the doubling of backslashes and the backslash before the default format (that is, control characters are represented by '`^C`' and meta characters as `M-C`). With this flag set, the encoding is ambiguous and non-invertible.

SEE ALSO

`unvis`(1), `vis`(1), `unvis`(3)

T. Berners-Lee, *Uniform Resource Locators (URL)*, RFC1738.

HISTORY

The *`vis`*, *`strvis`*, and *`strvisx`* functions first appeared in 4.4BSD. The *`svis`*, *`strsvis`*, and *`strsvisx`* functions appeared in NetBSD 1.5.

NAME

wcrtomb — converts a wide character to a multibyte character (restartable)

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

size_t
wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);
```

DESCRIPTION

wcrtomb() converts the wide character given by *wc* to the corresponding multibyte character, and stores it in the array pointed to by *s* unless *s* is a null pointer. This function will modify the first at most MB_CUR_MAX bytes of the array pointed to by *s*.

The behaviour of **wcrtomb**() is affected by the LC_CTYPE category of the current locale.

These are the special cases:

wc == 0 — For state-dependent encodings, **wcrtomb**() stores a nul byte preceded by special byte sequence (if any) to return to an initial state in the array pointed to by *s*, and the state object pointed to by *ps* also returns to an initial state.

s == NULL — **wcrtomb**() just places *ps* into an initial state. It is equivalent to the following call:

```
wcrtomb(buf, L'\0', ps);
```

Here, *buf* is a dummy buffer. In this case, *wc* is ignored.

ps == NULL — **mbrtowc**() uses its own internal state object to keep the conversion state, instead of *ps* mentioned in this manual page.

Calling any other functions in Standard C Library (libc, −lc) never changes the internal state of **mbrtowc**(), which is initialized at startup time of the program.

RETURN VALUES

wcrtomb() returns:

positive — The number of bytes (including any shift sequences) which are stored in the array.

(size_t)-1 — *wc* is not a valid wide character. In this case, **wcrtomb**() also sets *errno* to indicate the error.

ERRORS

wcrtomb() may cause an error in the following case:

[EILSEQ] — *wc* is not a valid wide character.

[EINVAL] — *ps* points to an invalid or uninitialized mbstate_t object.

SEE ALSO

setlocale(3), wctomb(3)

STANDARDS

The **wcrtomb**() function conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”). The restrict qualifier is added at ISO/IEC 9899:1999 (“ISO C99”).

NAME

wcscasecmp, **wcsncasecmp** — compare wide character strings, ignoring case

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

int
wcscasecmp(const wchar_t *s1, const wchar_t *s2);

int
wcsncasecmp(const wchar_t *s1, const wchar_t *s2, size_t len);
```

DESCRIPTION

The **wcscasecmp**() and **wcsncasecmp**() functions compare the nul-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* after translation of each corresponding character to lower-case. The strings themselves are not modified.

The **wcsncasecmp**() compares at most *len* characters.

SEE ALSO

wcscmp(3)

HISTORY

The **wcscasecmp**() and **wcsncasecmp**() functions first appeared in NetBSD 4.0.

NOTES

If *len* is zero, **wcsncasecmp**() returns always 0.

NAME

wcscoll — compare wide strings according to current collation

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

int
wcscoll(const wchar_t *s1, const wchar_t *s2);
```

DESCRIPTION

The **wcscoll**() function compares the nul-terminated strings *s1* and *s2* according to the current locale collation order. In the "C" locale, **wcscoll**() is equivalent to **wcscmp**().

RETURN VALUES

The **wcscoll**() function returns an integer greater than, equal to, or less than 0, if *s1* is greater than, equal to, or less than *s2*.

No return value is reserved to indicate errors; callers should set *errno* to 0 before calling **wcscoll**(). If it is non-zero upon return from **wcscoll**(), an error has occurred.

ERRORS

The **wcscoll**() function will fail if:

[EILSEQ] An invalid wide character code was specified.

[ENOMEM] Cannot allocate enough memory for temporary buffers.

SEE ALSO

setlocale(3), strcoll(3), wcscmp(3), wcsxfrm(3)

STANDARDS

The **wcscoll**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

BUGS

The current implementation of **wcscoll**() function disregards LC_COLLATE locales, and falls back to using the **wcscmp**() function.

NAME

wcsdup — save a copy of a string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

wchar_t *
wcsdup(const wchar_t *str);
```

DESCRIPTION

The **wcsdup**() function allocates sufficient memory for a copy of the wide character string *str*, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3).

If insufficient memory is available, NULL is returned.

EXAMPLES

The following will point *p* to an allocated area of memory containing the nul-terminated string "foobar":

```
wchar_t *p;

if (p = wcsdup(L"foobar"), p == NULL) {
        fprintf(stderr, "Out of memory.\n");
        exit(1);
}
```

ERRORS

The **wcsdup**() function may fail and set the external variable *errno* for any of the errors specified for the library function malloc(3).

SEE ALSO

free(3), malloc(3), strdup(3)

HISTORY

The **wcsdup**() function first appeared in NetBSD 4.0.

NAME

wcsftime — convert date and time to a wide-character string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

size_t
wcsftime(wchar_t * restrict wcs, size_t maxsize,
    const wchar_t * restrict format, const struct tm * restrict timeptr);
```

DESCRIPTION

The **wcsftime**() function is equivalent to the **strftime**() function except for the types of its arguments and the return value indicating the number of wide characters. Refer to strftime(3) for a detailed description.

COMPATIBILITY

Some early implementations of **wcsftime**() had a *format* argument with type *const char* * instead of *const wchar_t* *.

SEE ALSO

strftime(3)

STANDARDS

The **wcsftime**() function conforms to ISO/IEC 9899:1999 (“ISO C99”).

NAME

wcsrtombs — converts a wide character string to a multibyte character string (restartable)

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

size_t
wcsrtombs(char * restrict s, const wchar_t ** restrict pwcs, size_t n,
    mbstate_t * restrict ps);
```

DESCRIPTION

The **wcsrtombs**() converts the nul-terminated wide character string indirectly pointed to by *pwcs* to the corresponding multibyte character string, and stores it in the array pointed to by *s*. The conversion stops due to the following reasons:

- The conversion reaches a nul wide character. In this case, the nul wide character is also converted.
- The **wcsrtombs**() has already stored *n* bytes in the array pointed to by *s*.
- The conversion encounters an invalid character.

Each character will be converted as if wcrtomb(3) is continuously called, except the internal state of wcrtomb(3) will not be affected.

After conversion, if *s* is not a null pointer, the pointer object pointed to by *pwcs* is a null pointer (if the conversion is stopped due to reaching a nul wide character) or the first byte of the character just after the last character converted.

If *s* is not a null pointer and the conversion is stopped due to reaching a nul wide character, **wcsrtombs**() places the state object pointed to by *ps* to an initial state after the conversion is taken place.

The behaviour of **wcsrtombs**() is affected by the LC_CTYPE category of the current locale.

These are the special cases:

s == NULL
: **wcsrtombs**() returns the number of bytes to store the whole multibyte character string corresponding to the wide character string pointed to by *pwcs*, not including the terminating nul byte. In this case, *n* is ignored.

pwcs == NULL || *pwcs == NULL
: Undefined (may cause the program to crash).

ps == NULL
: **wcsrtombs**() uses its own internal state object to keep the conversion state, instead of *ps* mentioned in this manual page.

  Calling any other functions in Standard C Library (libc, −lc) never changes the internal state of **wcsrtombs**(), which is initialized at startup time of the program.

RETURN VALUES

wcsrtombs() returns:

0 or positive
: Number of bytes stored in the array pointed to by *s*, except for a nul byte. There are no cases that the value returned is greater than *n* (unless *s* is a null pointer). If the return value is equal to *n*, the string pointed to by *s* will not be nul-terminated.

(size_t)-1 *pwcs* points to a string containing an invalid wide character. The **wcsrtombs**() also sets *errno* to indicate the error.

ERRORS

wcsrtombs() may cause an error in the following case:

[EILSEQ] *pwcs* points to a string containing an invalid wide character.

SEE ALSO

setlocale(3), wcrtomb(3), wcstombs(3)

STANDARDS

The **wcsrtombs**() function conforms to ANSI X3.159-1989 ("ANSI C89"). The restrict qualifier is added at ISO/IEC 9899:1999 ("ISO C99").

NAME

wcstof, **wcstod**, **wcstold** — convert string to *float*, *double*, or *long double*

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

float
wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);

long double
wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);

double
wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

DESCRIPTION

The **wcstof**(), **wcstod**(), and **wcstold**() functions are the wide-character versions of the **strtof**(), **strtod**(), and **strtold**() functions. Refer to strtod(3) for details.

SEE ALSO

strtod(3), wcstol(3)

STANDARDS

The **wcstod**() function conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”). The **wcstof**() and **wcstold**() functions conform to ISO/IEC 9899:1999 (“ISO C99”).

NAME

wcstok — split wide-character string into tokens

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

wchar_t *
wcstok(wchar_t * restrict str, const wchar_t * restrict sep,
    wchar_t ** restrict last);
```

DESCRIPTION

The **wcstok**() function is used to isolate sequential tokens in a nul-terminated wide character string, *str*. These tokens are separated in the string by at least one of the characters in *sep*. The first time that **wcstok**() is called, *str* should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead. The separator string, *sep*, must be supplied each time, and may change between calls. The context pointer *last* must be provided on each call.

The **wcstok**() function is the wide character counterpart of the **strtok_r**() function.

RETURN VALUES

The **wcstok**() function returns a pointer to the beginning of each subsequent token in the string, after replacing the token itself with a nul wide character (L'\0'). When no more tokens remain, a null pointer is returned.

EXAMPLES

The following code fragment splits a wide character string on ASCII space, tab and newline characters and writes the tokens to standard output:

```
const wchar_t *seps = L" \t\n";
wchar_t *last, *tok, text[] = L" \none\ttwo\t\tthree  \n";

for (tok = wcstok(text, seps, &last); tok != NULL;
    tok = wcstok(NULL, seps, &last))
        wprintf(L"%ls\n", tok);
```

SEE ALSO

strtok(3), wcschr(3), wcscspn(3), wcspbrk(3), wcsrchr(3), wcsspn(3)

STANDARDS

The **wcstok**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

Some early implementations of **wcstok**() omit the context pointer argument, *last*, and maintain state across calls in a static variable like strtok(3) does.

NAME

wcstol, **wcstoul**, **wcstoll**, **wcstoull**, **wcstoimax**, **wcstoumax** — convert a wide character string value to a *long*, *unsigned long*, *long long*, *unsigned long long*, *intmax_t* or *uintmax_t* integer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

long
wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);

unsigned long
wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
    int base);

long long
wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
    int base);

unsigned long long
wcstoull(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
    int base);

#include <inttypes.h>

intmax_t
wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
    int base);

uintmax_t
wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
    int base);
```

DESCRIPTION

The **wcstol**(), **wcstoul**(), **wcstoll**(), **wcstoull**(), **wcstoimax**() and **wcstoumax**() functions are wide-character versions of the **strtol**(), **strtoul**(), **strtoll**(), **strtoull**(), **strtoimax**() and **strtoumax**() functions, respectively. Refer to their manual pages (for example strtol(3)) for details.

SEE ALSO

strtol(3), strtoul(3)

STANDARDS

The **wcstol**(), **wcstoul**(), **wcstoll**(), **wcstoull**(), **wcstoimax**() and **wcstoumax**() functions conform to ISO/IEC 9899:1999 (“ISO C99”).

NAME

wcstombs — converts a wide character string to a multibyte character string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

size_t
wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);
```

DESCRIPTION

wcstombs() converts the nul-terminated wide character string pointed to by *pwcs* to the corresponding multibyte character string, and stores it in the array pointed to by *s*. This function may modify the first at most *n* bytes of the array pointed to by *s*. Each character will be converted as if wctomb(3) is continuously called, except the internal state of wctomb(3) will not be affected.

For state-dependent encoding, the **wcstombs**() implies the result multibyte character string pointed to by *s* always to begin with an initial state.

The behaviour of **wcstombs**() is affected by the LC_CTYPE category of the current locale.

These are the special cases:

s == NULL — The **wcstombs**() returns the number of bytes to store the whole multibyte character string corresponding to the wide character string pointed to by *pwcs*. In this case, *n* is ignored.

pwcs == NULL — Undefined (may cause the program to crash).

RETURN VALUES

wcstombs() returns:

0 or positive — Number of bytes stored in the array pointed to by *s*. There are no cases that the value returned is greater than *n* (unless *s* is a null pointer). If the return value is equal to *n*, the string pointed to by *s* will not be nul-terminated.

(size_t)-1 — *pwcs* points to a string containing an invalid wide character. **wcstombs**() also sets *errno* to indicate the error.

ERRORS

wcstombs() may cause an error in the following case:

[EILSEQ] — *pwcs* points to a string containing an invalid wide character.

SEE ALSO

setlocale(3), wctomb(3)

STANDARDS

The **wcstombs**() function conforms to ANSI X3.159-1989 (“ANSI C89”). The restrict qualifier is added at ISO/IEC 9899:1999 (“ISO C99”).

NAME

wcswidth — number of column positions in wide-character string

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

int
wcswidth(const wchar_t *pwcs, size_t n);
```

DESCRIPTION

The **wcswidth**() function determines the number of column positions required for the first *n* characters of *pwcs*, or until a nul wide character (L'\0') is encountered.

RETURN VALUES

The **wcswidth**() function returns 0 if *pwcs* is an empty string (L""), −1 if a non-printing wide character is encountered, otherwise it returns the number of column positions occupied.

SEE ALSO

iswprint(3), wcwidth(3)

STANDARDS

The **wcswidth**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

wcsxfrm — transform a wide string under locale

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

size_t
wcsxfrm(wchar_t * restrict dst, const wchar_t * restrict src, size_t n);
```

DESCRIPTION

The **wcsxfrm**() function transforms a nul-terminated wide character string pointed to by *src* according to the current locale collation order then copies the transformed string into *dst*. No more than *n* wide characters are copied into *dst*, including the terminating nul character added. If *n* is set to 0 (it helps to determine an actual size needed for transformation), *dst* is permitted to be a null pointer.

Comparing two strings using **wcscmp**() after **wcsxfrm**() is equivalent to comparing two original strings with **wcscoll**().

RETURN VALUES

Upon successful completion, **wcsxfrm**() returns the length of the transformed string not including the terminating nul character. If this value is *n* or more, the contents of *dst* are indeterminate.

SEE ALSO

setlocale(3), strxfrm(3), wcscmp(3), wcscoll(3)

STANDARDS

The **wcsxfrm**() function conforms to ISO/IEC 9899:1999 (“ISO C99”).

BUGS

The current implementation of **wcsxfrm**() function disregards LC_COLLATE locales, and falls back to using the **wcsncpy**() function.

NAME

wctob — convert a wide character to a single byte character

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

int
wctob(wint_t wc);
```

DESCRIPTION

The **wctob**() function converts a wide character *wc* to a corresponding single byte character in the initial shift state of the current locale.

The behaviour of **wctob**() is affected by the LC_CTYPE category of the current locale.

RETURN VALUES

The **wctob**() function returns:

EOF	If *wc* is WEOF or if *wc* does not correspond to a valid single byte character representation.
(otherwise)	A single byte character corresponding to *wc*.

ERRORS

No errors are defined.

SEE ALSO

btowc(3), setlocale(3), wcrtomb(3)

STANDARDS

The **wctob**() function conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”).

NAME

wctomb — converts a wide character to a multibyte character

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdlib.h>

int
wctomb(char * s, const wchar_t wchar);
```

DESCRIPTION

The **wctomb**() converts the wide character *wchar* to the corresponding multibyte character, and stores it in the array pointed to by *s*. **wctomb**() may store at most MB_CUR_MAX bytes in the array.

In state-dependent encoding, **wctomb**() may store the special sequence to change the conversion state before an actual multibyte character into the array pointed to by *s*. If *wchar* is a nul wide character ('\0'), this function sets its own internal state to an initial conversion state.

Calling any other functions in Standard C Library (libc, −lc) never changes the internal state of **wctomb**(), except changing the LC_CTYPE category of the current locale by calling setlocale(3). Such setlocale(3) calls cause the internal state of this function to be indeterminate.

The behaviour of **wctomb**() is affected by the LC_CTYPE category of the current locale.

There is one special case:

s == NULL — **wctomb**() initializes its own internal state to an initial state, and determines whether the current encoding is state-dependent. This function returns 0 if the encoding is state-independent, otherwise non-zero. In this case, *wchar* is completely ignored.

RETURN VALUES

Normally, **wctomb**() returns:

positive — Number of bytes for the valid multibyte character pointed to by *s*. There are no cases that the value returned is greater than *n* or the value of the MB_CUR_MAX macro.

-1 — *wchar* is an invalid wide character.

If *s* is equal to NULL, **mbtowc**() returns:

0 — The current encoding is state-independent.

non-zero — The current encoding is state-dependent.

ERRORS

No errors are defined.

SEE ALSO

setlocale(3)

STANDARDS

The **wctomb**() function conforms to ANSI X3.159-1989 ("ANSI C89").

NAME

wctrans — get character mapping identifier by name

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wctype.h>

wctrans_t
wctrans(const char *charmap);
```

DESCRIPTION

The **wctrans**() function returns a character mapping identifier corresponding to the locale-specific character mapping name *charmap*. This identifier can be used on the subsequent calls of **towctrans**(). The following names are defined in all locales:

```
tolower toupper
```

The behaviour of **wctrans**() is affected by the LC_CTYPE category of the current locale.

RETURN VALUES

wctrans() returns:

0	If the string *charmap* does not corresponding to a valid character mapping name.
non-zero	A character mapping identifier corresponding to *charmap*.

Note: wctype_t is a scalar type, e.g., a pointer.

ERRORS

No errors are defined.

SEE ALSO

iswctype(3), setlocale(3), wctype(3)

STANDARDS

The **towctrans**() function conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”).

NAME

wctype — get character class identifier by name

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wctype.h>

wctype_t
wctype(const char *charclass);
```

DESCRIPTION

The **wctype**() function returns a character class identifier corresponding to the locale-specific character class name *charclass*. This identifier can be used in subsequent calls of **iswctype**(). The following names are defined in all locales:

```
alnum alpha blank cntrl digit graph
lower print punct space upper xdigit
```

The behaviour of **wctype**() is affected by the LC_CTYPE category of the current locale.

RETURN VALUES

wctype() returns:

0	If *charclass* does not correspond to a valid character class name.
non-zero	A character class identifier corresponding to *charclass*.

Note: wctype_t is a scalar type, e.g., a pointer.

ERRORS

No errors are defined.

SEE ALSO

iswctype(3), setlocale(3), towctrans(3), wctrans(3)

STANDARDS

The **wctype**() function conforms to ISO/IEC 9899/AMD1:1995 (“ISO C90, Amendment 1”).

NAME

wcwidth — number of column positions of a wide-character code

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

int
wcwidth(wchar_t wc);
```

DESCRIPTION

The **wcwidth**() function determines the number of column positions required to display the wide character *wc*.

RETURN VALUES

The **wcwidth**() function returns 0 if the *wc* argument is a nul wide character (L'\0'), −1 if *wc* is not printable, otherwise it returns the number of column positions the character occupies.

EXAMPLES

This code fragment reads text from standard input and breaks lines that are more than 20 column positions wide, similar to the `fold`(1) utility:

```
wint_t ch;
int column, w;

column = 0;
while ((ch = getwchar()) != WEOF) {
        w = wcwidth(ch);
        if (w > 0 && column + w >= 20) {
                putwchar(L'\n');
                column = 0;
        }
        putwchar(ch);
        if (ch == L'\n')
                column = 0;
        else if (w > 0)
                column += w;
}
```

SEE ALSO

`iswprint`(3), `wcswidth`(3)

STANDARDS

The **wcwidth**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

wmemchr, **wmemcmp**, **wmemcpy**, **wmemmove**, **wmemset**, **wcscat**, **wcschr**, **wcscmp**, **wcscpy**, **wcscspn**, **wcslcat**, **wcslcpy**, **wcslen**, **wcsncat**, **wcsncmp**, **wcsncpy**, **wcspbrk**, **wcsrchr**, **wcsspn**, **wcsstr** **wcswcs** — wide character string manipulation operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wchar.h>

wchar_t *
wmemchr(const wchar_t *s, wchar_t c, size_t n);

int
wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);

wchar_t *
wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);

wchar_t *
wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);

wchar_t *
wmemset(wchar_t *s, wchar_t c, size_t n);

wchar_t *
wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);

wchar_t *
wcschr(const wchar_t *s, wchar_t c);

int
wcscmp(const wchar_t *s1, const wchar_t *s2);

wchar_t *
wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);

size_t
wcscspn(const wchar_t *s1, const wchar_t *s2);

size_t
wcslcat(wchar_t *s1, const wchar_t *s2, size_t n);

size_t
wcslcpy(wchar_t *s1, const wchar_t *s2, size_t n);

size_t
wcslen(const wchar_t *s);

wchar_t *
wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);

int
wcsncmp(const wchar_t *s1, const wchar_t * s2, size_t n);

wchar_t *
wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

wchar_t *
wcspbrk(*const wchar_t *s1*, *const wchar_t *s2*);

wchar_t *
wcsrchr(*const wchar_t *s*, *wchar_t c*);

size_t
wcsspn(*const wchar_t *s1*, *const wchar_t *s2*);

wchar_t *
wcsstr(*const wchar_t *s1*, *const wchar_t *s2*);

wchar_t *
wcswcs(*const wchar_t *s1*, *const wchar_t *s2*);

DESCRIPTION

These functions implement string manipulation operations over wide character strings. For a detailed description, refer to the documents for the respective single-byte counterpart, such as memchr(3). The **wcswcs**() function is not a part of ISO/IEC 9899:1990 ("ISO C90") and ISO/IEC 9899/AMD1:1995 ("ISO C90, Amendment 1"), the **wcsstr**() function is strongly recommended to be used.

SEE ALSO

memchr(3), memcmp(3), memcpy(3), memmove(3), memset(3), strcat(3), strchr(3), strcmp(3), strcpy(3), strcspn(3), strlcat(3), strlcpy(3), strlen(3), strncat(3), strncmp(3), strncpy(3), strpbrk(3), strrchr(3), strspn(3), strstr(3)

STANDARDS

These functions conform to ISO/IEC 9899:1999 ("ISO C99") and were first introduced in ISO/IEC 9899/AMD1:1995 ("ISO C90, Amendment 1"), with the exception of **wcslcat**() and **wcslcpy**(), which are extensions. The **wcswcs**() function conforms to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

NAME

wordexp — perform shell-style word expansions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <wordexp.h>

int
wordexp(const char * restrict words, wordexp_t * restrict pwordexp,
    int flags);

void
wordfree(wordexp_t *pwordexp);
```

DESCRIPTION

The **wordexp**() function performs shell-style word expansion on *words* and places the list of expanded words into the structure pointed to by *pwordexp*.

The *flags* argument is the bitwise inclusive OR of any of the following constants:

WRDE_APPEND Append the words to those generated by a previous call to **wordexp**().

WRDE_DOOFFS As many NULL pointers as are specified by the *we_offs* member of *we* are added to the front of *we_wordv*.

WRDE_NOCMD Disallow command substitution in *words*. See the note in **BUGS** before using this.

WRDE_REUSE The *we* argument was passed to a previous successful call to **wordexp**() but has not been passed to **wordfree**(). The implementation may reuse the space allocated to it.

WRDE_SHOWERR Do not redirect shell error messages to /dev/null.

WRDE_UNDEF Report error on an attempt to expand an undefined shell variable.

The structure type **wordexp_t** includes the following members:

```
size_t  we_wordc
char    **we_wordv
size_t  we_offs
```

The *we_wordc* member is the count of generated words.

The *we_wordv* member points to a list of pointers to expanded words.

The *we_offs* member is the number of slots to reserve at the beginning of the *we_wordv* member.

It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. The **wordexp**() function allocates other space as needed, including memory pointed to by the *we_wordv* member.

The **wordfree**() function frees the memory allocated by **wordexp**().

IMPLEMENTATION NOTES

The **wordexp**() function is implemented as a wrapper around the undocumented **wordexp** shell built-in command.

RETURN VALUES

The **wordexp**() function returns zero if successful, otherwise it returns one of the following error codes:

WRDE_BADCHAR The *words* argument contains one of the following unquoted characters: ⟨newline⟩, '|', '&', ';', '<', '>', '(', ')', '{', '}'.

WRDE_BADVAL An attempt was made to expand an undefined shell variable and WRDE_UNDEF is set in *flags*.

WRDE_CMDSUB An attempt was made to use command substitution and WRDE_NOCMD is set in *flags*.

WRDE_NOSPACE Not enough memory to store the result.

WRDE_SYNTAX Shell syntax error in *words*.

WRDE_ERRNO An internal error occured and *errno* is set to indicate the error.

The **wordfree**() function returns no value.

ENVIRONMENT

IFS Field separator.

EXAMPLES

Invoke the editor on all .c files in the current directory and /etc/motd (error checking omitted):

```
wordexp_t we;

wordexp("${EDITOR:-vi} *.c /etc/motd", &we, 0);
execvp(we->we_wordv[0], we->we_wordv);
```

DIAGNOSTICS

Diagnostic messages from the shell are written to the standard error output if WRDE_SHOWERR is set in *flags*.

SEE ALSO

sh(1), fnmatch(3), glob(3), popen(3), system(3)

STANDARDS

The **wordexp**() and **wordfree**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1"). Their first release was in IEEE Std 1003.2-1992 ("POSIX.2"). The return value WRDE_ERRNO is an extension.

BUGS

Do not pass untrusted user data to **wordexp**(), regardless of whether the WRDE_NOCMD flag is set. The **wordexp**() function attempts to detect input that would cause commands to be executed before passing it to the shell but it does not use the same parser so it may be fooled.

NAME

wprintf, **fwprintf**, **swprintf**, **vwprintf**, **vfwprintf**, **vswprintf** — formatted wide character output conversion

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int
fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);

int
swprintf(wchar_t * restrict ws, size_t n, const wchar_t * restrict format,
    ...);

int
wprintf(const wchar_t * restrict format, ...);

#include <stdarg.h>

int
vfwprintf(FILE * restrict stream, const wchar_t * restrict, va_list ap);

int
vswprintf(wchar_t * restrict ws, size_t n, const wchar_t *restrict format,
    va_list ap);

int
vwprintf(const wchar_t * restrict format, va_list ap);
```

DESCRIPTION

The **wprintf**() family of functions produces output according to a *format* as described below. The **wprintf**() and **vwprintf**() functions write output to `stdout`, the standard output stream; **fwprintf**() and **vfwprintf**() write output to the given output *stream*; **swprintf**() and **vswprintf**() write to the wide character string *ws*.

These functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of `stdarg`(3)) are converted for output.

These functions return the number of characters printed (not including the trailing '`\0`' used to end output to strings).

The **swprintf**() and **vswprintf**() functions will fail if *n* or more wide characters were requested to be written,

The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the **%** character. The arguments must correspond properly (after type promotion) with the conversion specifier. After the **%**, the following appear in sequence:

- An optional field, consisting of a decimal digit string followed by a **$**, specifying the next argument to access. If this field is not provided, the argument following the last argument accessed will be used. Arguments are numbered starting at **1**. If unaccessed arguments in the format string are interspersed

with ones that are accessed the results will be indeterminate.

- Zero or more of the following flags:

  '**#**' The value should be converted to an "alternate form". For **c**, **d**, **i**, **n**, **p**, **s**, and **u** conversions, this option has no effect. For **o** conversions, the precision of the number is increased to force the first character of the output string to a zero (except if a zero value is printed with an explicit precision of zero). For **x** and **X** conversions, a non-zero result has the string '0x' (or '0X' for **X** conversions) prepended to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For **g** and **G** conversions, trailing zeros are not removed from the result as they would otherwise be.

  '**0**' (zero) Zero padding. For all conversions except **n**, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (**d**, **i**, **o**, **u**, **i**, **x**, and **X**), the **0** flag is ignored.

  '**-**' A negative field width flag; the converted value is to be left adjusted on the field boundary. Except for **n** conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A **-** overrides a **0** if both are given.

  ' ' (space) A blank should be left before a positive number produced by a signed conversion (**a**, **A**, **d**, **e**, **E**, **f**, **F**, **g**, **G**, or **i**).

  '**+**' A sign must always be placed before a number produced by a signed conversion. A **+** overrides a space if both are used.

  ''' Decimal conversions (**d**, **u**, or **i**) or the integral portion of a floating point conversion (**f** or **F**) should be grouped and separated by thousands using the non-monetary separator returned by `localeconv`(3).

- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given) to fill out the field width.

- An optional precision, in the form of a period **.** followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear for **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for **g** and **G** conversions, or the maximum number of characters to be printed from a string for **s** conversions.

- An optional length modifier, that specifies the size of the argument. The following length modifiers are valid for the **d**, **i**, **n**, **o**, **u**, **x**, or **X** conversion:

Modifier	**d**, **i**	**o**, **u**, **x**, **X**	**n**
hh	*signed char*	*unsigned char*	*signed char* *
h	*short*	*unsigned short*	*short* *
l (ell)	*long*	*unsigned long*	*long* *
ll (ell ell)	*long long*	*unsigned long long*	*long long* *
j	*intmax_t*	*uintmax_t*	*intmax_t* *
t	*ptrdiff_t*	(see note)	*ptrdiff_t* *
z	(see note)	*size_t*	(see note)
q *(deprecated)*	*quad_t*	*u_quad_t*	*quad_t* *

  Note: the **t** modifier, when applied to a **o**, **u**, **x**, or **X** conversion, indicates that the argument is of an unsigned type equivalent in size to a *ptrdiff_t*. The **z** modifier, when applied to a **d** or **i** conversion, indicates that the argument is of a signed type equivalent in size to a *size_t*. Similarly, when applied to an **n** conversion, it indicates that the argument is a pointer to a signed type equivalent in size to a

size_t.

The following length modifier is valid for the **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion:

Modifier	**a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**
L	*long double*

The following length modifier is valid for the **c** or **s** conversion:

Modifier	**c**	**s**
l (ell)	*wint_t*	*wchar_t* *

- A character that specifies the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk '*' or an asterisk followed by one or more decimal digits and a '$' instead of a digit string. In this case, an *int* argument supplies the field width or precision. A negative field width is treated as a left adjustment flag followed by a positive field width; a negative precision is treated as though it were missing. If a single format directive mixes positional (nn$) and non-positional arguments, the results are undefined.

The conversion specifiers and their meanings are:

diouxX The *int* (or appropriate variant) argument is converted to signed decimal (**d** and **i**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation. The letters "abcdef" are used for **x** conversions; the letters "ABCDEF" are used for **X** conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros.

DOU The *long int* argument is converted to signed decimal, unsigned octal, or unsigned decimal, as if the format had been **ld**, **lo**, or **lu** respectively. These conversion characters are deprecated, and will eventually disappear.

eE The *double* argument is rounded and converted in the style [−]*d*.*ddd*e±*dd* where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An **E** conversion uses the letter 'E' (rather than 'e') to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, positive and negative infinity are represented as inf and -inf respectively when using the lowercase conversion character, and INF and -INF respectively when using the uppercase conversion character. Similarly, NaN is represented as nan when using the lowercase conversion, and NAN when using the uppercase conversion.

fF The *double* argument is rounded and converted to decimal notation in the style [−]*ddd*.*ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

gG The *double* argument is converted in style **f** or **e** (or **F** or **E** for **G** conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style **e** is used if the exponent from its conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

aA The *double* argument is converted to hexadecimal notation in the style [−]0x*h*.*hhhp*[±]*d*, where the number of digits after the hexadecimal-point character is equal to the precision specification. If the precision is missing, it is taken as enough to exactly represent the floating-point number; if the precision is explicitly zero, no hexadecimal-point character appears. This is an exact conversion of the mantissa+exponent internal floating point representation; the [−]0x*h*.*hhh*

portion represents exactly the mantissa; only denormalized mantissas have a zero value to the left of the hexadecimal point. The **p** is a literal character 'p'; the exponent is preceded by a positive or negative sign and is represented in decimal, using only enough characters to represent the exponent. The **A** conversion uses the prefix "0X" (rather than "0x"), the letters "ABCDEF" (rather than "abcdef") to represent the hex digits, and the letter 'P' (rather than 'p') to separate the mantissa and exponent.

C Treated as **c** with the **l** (ell) modifier.

c The *int* argument is converted to an *unsigned char*, then to a *wchar_t* as if by btowc(3), and the resulting character is written.

If the **l** (ell) modifier is used, the *wint_t* argument is converted to a *wchar_t* and written.

S Treated as **s** with the **l** (ell) modifier.

s The *char* * argument is expected to be a pointer to an array of character type (pointer to a string) containing a multibyte sequence. Characters from the array are converted to wide characters and written up to (but not including) a terminating NUL character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating NUL character.

If the **l** (ell) modifier is used, the *wchar_t* * argument is expected to be a pointer to an array of wide characters (pointer to a wide string). Each wide character in the string is written. Wide characters from the array are written up to (but not including) a terminating wide NUL character; if a precision is specified, no more than the number specified are written (including shift sequences). If a precision is given, no null character need be present; if the precision is not specified, or is greater than the number of characters in the string, the array must contain a terminating wide NUL character.

p The *void* * pointer argument is printed in hexadecimal (as if by %#x or %#lx).

n The number of characters written so far is stored into the integer indicated by the *int* * (or variant) pointer argument. No argument is converted.

% A '%' is written. No argument is converted. The complete conversion specification is '%%'.

The decimal point character is defined in the program's locale (category LC_NUMERIC).

In no case does a non-existent or small field width cause truncation of a numeric field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

SEE ALSO

btowc(3), fputws(3), printf(3), putwc(3), setlocale(3), wcsrtombs(3), wscanf(3)

STANDARDS

Subject to the caveats noted in the

SECURITY CONSIDERATIONS

Refer to printf(3). **BUGS** section of printf(3), the **wprintf**(), **fwprintf**(), **swprintf**(), **vwprintf**(), **vfwprintf**() and **vswprintf**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

NAME

wscanf, **fwscanf**, **swscanf**, **vwscanf**, **vswscanf**, **vfwscanf** — wide character input format conversion

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int
wscanf(const wchar_t * restrict format, ...);

int
fwscanf(FILE * restrict stream, const wchar_t * restrict format, ...);

int
swscanf(const wchar_t * restrict str, const wchar_t * restrict format,
    ...);

#include <stdarg.h>

int
vwscanf(const wchar_t * restrict format, va_list ap);

int
vswscanf(const wchar_t * restrict str, const wchar_t * restrict format,
    va_list ap);

int
vfwscanf(FILE * restrict stream, const wchar_t * restrict format,
    va_list ap);
```

DESCRIPTION

The **wscanf**() family of functions scans input according to a *format* as described below. This format may contain *conversion specifiers*; the results from such conversions, if any, are stored through the *pointer* arguments. The **wscanf**() function reads input from the standard input stream `stdin`, **fwscanf**() reads input from the stream pointer *stream*, and **swscanf**() reads its input from the wide character string pointed to by *str*. The **vfwscanf**() function is analogous to `vfwprintf`(3) and reads input from the stream pointer *stream* using a variable argument list of pointers (see `stdarg`(3)). The **vwscanf**() function scans a variable argument list from the standard input and the **vswscanf**() function scans it from a wide character string; these are analogous to the **vwprintf**() and **vswprintf**() functions respectively. Each successive *pointer* argument must correspond properly with each successive conversion specifier (but see the * conversion below). All conversions are introduced by the **%** (percent sign) character. The *format* string may also contain other characters. White space (such as blanks, tabs, or newlines) in the *format* string match any amount of white space, including none, in the input. Everything else matches only itself. Scanning stops when an input character does not match such a format character. Scanning also stops when an input conversion cannot be made (see below).

CONVERSIONS

Following the **%** character introducing a conversion there may be a number of *flag* characters, as follows:

* Suppresses assignment. The conversion that follows occurs as usual, but no pointer is used; the result of the conversion is simply discarded.

hh Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *char* (rather than *int*).

h Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *short int* (rather than *int*).

l (ell) Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *long int* (rather than *int*), that the conversion will be one of **a**, **e**, **f**, or **g** and the next pointer is a pointer to *double* (rather than *float*), or that the conversion will be one of **c** or **s** and the next pointer is a pointer to an array of *wchar_t* (rather than *char*).

ll (ell ell)
Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *long long int* (rather than *int*).

L Indicates that the conversion will be one of **a**, **e**, **f**, or **g** and the next pointer is a pointer to *long double*.

j Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *intmax_t* (rather than *int*).

t Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *ptrdiff_t* (rather than *int*).

z Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *size_t* (rather than *int*).

q (deprecated.) Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *long long int* (rather than *int*).

In addition to these flags, there may be an optional maximum field width, expressed as a decimal integer, between the **%** and the conversion. If no width is given, a default of "infinity" is used (with one exception, below); otherwise at most this many characters are scanned in processing the conversion. Before conversion begins, most conversions skip white space; this white space is not counted against the field width.

The following conversions are available:

% Matches a literal '%'. That is, "%%" in the format string matches a single input '%' character. No conversion is done, and assignment does not occur.

d Matches an optionally signed decimal integer; the next pointer must be a pointer to *int*.

i Matches an optionally signed integer; the next pointer must be a pointer to *int*. The integer is read in base 16 if it begins with '0x' or '0X', in base 8 if it begins with '0', and in base 10 otherwise. Only characters that correspond to the base are used.

o Matches an octal integer; the next pointer must be a pointer to *unsigned int*.

u Matches an optionally signed decimal integer; the next pointer must be a pointer to *unsigned int*.

x, **X** Matches an optionally signed hexadecimal integer; the next pointer must be a pointer to *unsigned int*.

a, **A**, **e**, **E**, **f**, **F**, **g**, **G**
Matches a floating-point number in the style of wcstod(3). The next pointer must be a pointer to *float* (unless **l** or **L** is specified.)

s Matches a sequence of non-white-space wide characters; the next pointer must be a pointer to *char*, and the array must be large enough to accept the multibyte representation of all the sequence and the terminating NUL character. The input string stops at white space or at the maximum field width, whichever occurs first.

If an **l** qualifier is present, the next pointer must be a pointer to *wchar_t*, into which the input will be placed.

S The same as **ls**.

c Matches a sequence of *width* count wide characters (default 1); the next pointer must be a pointer to *char*, and there must be enough room for the multibyte representation of all the characters (no terminating NUL is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.

If an **l** qualifier is present, the next pointer must be a pointer to *wchar_t*, into which the input will be placed.

C The same as **lc**.

[Matches a nonempty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to *char*, and there must be enough room for the multibyte representation of all the characters in the string, plus a terminating NUL character. The usual skip of leading white space is suppressed. The string is to be made up of characters in (or not in) a particular set; the set is defined by the characters between the open bracket [character and a close bracket] character. The set *excludes* those characters if the first character after the open bracket is a circumflex ^. To include a close bracket in the set, make it the first character after the open bracket or the circumflex; any other position will end the set. To include a hyphen in the set, make it the last character before the final close bracket; some implementations of **wscanf**() use "A-Z" to represent the range of characters between 'A' and 'Z'. The string ends with the appearance of a character not in the (or, with a circumflex, in) set or when the field width runs out.

If an **l** qualifier is present, the next pointer must be a pointer to *wchar_t*, into which the input will be placed.

p Matches a pointer value (as printed by '%p' in wprintf(3)); the next pointer must be a pointer to *void*.

n Nothing is expected; instead, the number of characters consumed thus far from the input is stored through the next pointer, which must be a pointer to *int*. This is *not* a conversion, although it can be suppressed with the * flag.

The decimal point character is defined in the program's locale (category LC_NUMERIC).

For backwards compatibility, a "conversion" of %\0 causes an immediate return of EOF.

RETURN VALUES

These functions return the number of input items assigned, which can be fewer than provided for, or even zero, in the event of a matching failure. Zero indicates that, while there was input available, no conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a '%d' conversion. The value EOF is returned if an input failure occurs before any conversion such as an end-of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned.

SEE ALSO

fgetwc(3), scanf(3), wcrtomb(3), wcstod(3), wcstol(3), wcstoul(3), wprintf(3)

STANDARDS

The **fwscanf**(), **wscanf**(), **swscanf**(), **vfwscanf**(), **vwscanf**() and **vswscanf**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

BUGS

In addition to the bugs documented in `scanf`(3), **`wscanf`**() does not support the "`A-Z`" notation for specifying character ranges with the character class conversion ('**`%[`**').

NAME

xdr, **xdr_array**, **xdr_bool**, **xdr_bytes**, **xdr_char**, **xdr_destroy**, **xdr_double**, **xdr_enum**, **xdr_float**, **xdr_free**, **xdr_getpos**, **xdr_hyper**, **xdr_inline**, **xdr_int**, **xdr_long**, **xdr_longlong_t**, **xdrmem_create**, **xdr_opaque**, **xdr_pointer**, **xdrrec_create**, **xdrrec_endofrecord**, **xdrrec_eof**, **xdrrec_skiprecord**, **xdr_reference**, **xdr_setpos**, **xdr_short**, **xdrstdio_create**, **xdr_string**, **xdr_u_char**, **xdr_u_hyper**, **xdr_u_long**, **xdr_u_longlong_t**, **xdr_u_short**, **xdr_union**, **xdr_vector**, **xdr_void**, **xdr_wrapstring** — library routines for external data representation

SYNOPSIS

int
xdr_array(*XDR *xdrs*, *char **arrp*, *u_int *sizep*, *u_int maxsize*, *u_int elsize*, *xdrproc_t elproc*);

int
xdr_bool(*XDR *xdrs*, *bool_t *bp*);

int
xdr_bytes(*XDR *xdrs*, *char **sp*, *u_int *sizep*, *u_int maxsize*);

int
xdr_char(*XDR *xdrs*, *char *cp*);

void
xdr_destroy(*XDR *xdrs*);

int
xdr_double(*XDR *xdrs*, *double *dp*);

int
xdr_enum(*XDR *xdrs*, *enum_t *ep*);

int
xdr_float(*XDR *xdrs*, *float *fp*);

void
xdr_free(*xdrproc_t proc*, *char *objp*);

u_int
xdr_getpos(*XDR *xdrs*);

int
xdr_hyper(*XDR *xdrs*, *longlong_t *llp*);

*long **
xdr_inline(*XDR *xdrs*, *int len*);

int
xdr_int(*XDR *xdrs*, *int *ip*);

int
xdr_long(*XDR *xdrs*, *long *lp*);

int
xdr_longlong_t(*XDR *xdrs*, *longlong_t *llp*);

void
xdrmem_create(*XDR *xdrs*, *char *addr*, *u_int size*, *enum xdr_op op*);

int
xdr_opaque(*XDR *xdrs*, *char *cp*, *u_int cnt*);

int
xdr_pointer(*XDR *xdrs*, *char **objpp*, *u_int objsize*, *xdrproc_t xdrobj*);

void
xdrrec_create(*XDR *xdrs*, *u_int sendsize*, *u_int recvsize*, *char *handle*, *int (*readit)()*, *int (*writeit)()*);

int
xdrrec_endofrecord(*XDR *xdrs*, *int sendnow*);

int
xdrrec_eof(*XDR *xdrs*);

int
xdrrec_skiprecord(*XDR *xdrs*);

int
xdr_reference(*XDR *xdrs*, *char **pp*, *u_int size*, *xdrproc_t proc*);

int
xdr_setpos(*XDR *xdrs*, *u_int pos*);

int
xdr_short(*XDR *xdrs*, *short *sp*);

void
xdrstdio_create(*XDR *xdrs*, *FILE *file*, *enum xdr_op op*);

int
xdr_string(*XDR *xdrs*, *char **sp*, *u_int maxsize*);

int
xdr_u_char(*XDR *xdrs*, *unsigned char *ucp*);

int
xdr_u_hyper(*XDR *xdrs*, *u_longlong_t *ullp*);

int
xdr_u_int(*XDR *xdrs*, *unsigned *up*);

int
xdr_u_long(*XDR *xdrs*, *unsigned long *ulp*);

int
xdr_u_longlong_t(*XDR *xdrs*, *u_longlong_t *ullp*);

int
xdr_u_short(*XDR *xdrs*, *unsigned short *usp*);

int
xdr_union(*XDR *xdrs*, *int *dscmp*, *char *unp*, *struct xdr_discrim *choices*, *bool_t (*defaultarm)()*);

int
xdr_vector(*XDR *xdrs*, *char *arrp*, *u_int size*, *u_int elsize*, *xdrproc_t elproc*);

int
xdr_void(*void*);

int
xdr_wrapstring(*XDR *xdrs*, *char **sp*);

DESCRIPTION

These routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls are transmitted using these routines.

xdr_array()
A filter primitive that translates between variable-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *sizep* is the address of the element count of the array; this element count cannot exceed *maxsize*. The parameter *elsize* is the *sizeof* each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

xdr_bool()
A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

xdr_bytes()
A filter primitive that translates between counted byte strings and their external representations. The parameter *sp* is the address of the string pointer. The length of the string is located at address *sizep*; strings cannot be longer than *maxsize*. This routine returns one if it succeeds, zero otherwise.

xdr_char()
A filter primitive that translates between C characters and their external representations. This routine returns one if it succeeds, zero otherwise. Note: encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider **xdr_bytes**(), **xdr_opaque**() or **xdr_string**().

xdr_destroy()
A macro that invokes the destroy routine associated with the XDR stream, *xdrs*. Destruction usually involves freeing private data structures associated with the stream. Using *xdrs* after invoking **xdr_destroy**() is undefined.

xdr_double()
A filter primitive that translates between C double precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_enum()
A filter primitive that translates between C enums (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_float()
A filter primitive that translates between C floats and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_free()
Generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself. Note: the pointer passed to this routine is *not* freed, but what it points to *is* freed (recursively).

xdr_getpos()
A macro that invokes the get-position routine associated with the XDR stream, *xdrs*. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

xdr_hyper()
A filter primitive that translates between ANSI C long long integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_inline()
A macro that invokes the in-line routine associated with the XDR stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note: pointer is cast to *long* *.

Warning: **xdr_inline**() may return NULL if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

xdr_int()
A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_long()
A filter primitive that translates between C long integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_longlong_t()
A filter primitive that translates between ANSI C long long integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdrmem_create()
This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to, or read from, a chunk of memory at location *addr* whose length is no more than *size* bytes long. The *op* determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

xdr_opaque()
A filter primitive that translates between fixed size opaque data and its external representation. The parameter *cp* is the address of the opaque object, and *cnt* is its size in bytes. This routine returns one if it succeeds, zero otherwise.

xdr_pointer()
Like **xdr_reference**() except that it serializes NULL pointers, whereas **xdr_reference**() does not. Thus, **xdr_pointer**() can represent recursive data structures, such as binary trees or linked lists.

xdrrec_create()
This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to a buffer of size *sendsize*; a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsize*; it too can be set to a suitable default by passing a zero value. When a stream's output buffer is full, *writeit* is called. Similarly, when a stream's input buffer is empty, *readit* is called. The behavior of these two routines is similar to the system calls read(2) and write(2), except that *handle* is passed to the former routines as the first parameter. Note: the XDR stream's *op* field must be set by the caller.

Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

xdrrec_endofrecord()
This routine can be invoked only on streams created by **xdrrec_create**(). The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if *sendnow* is non-zero. This routine returns one if it succeeds, zero otherwise.

xdrrec_eof()
This routine can be invoked only on streams created by **xdrrec_create**(). After consuming the rest of the current record in the stream, this routine returns one if the stream has no more input, zero otherwise.

xdrrec_skiprecord()
This routine can be invoked only on streams created by **xdrrec_create**(). It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns one if it succeeds, zero otherwise.

xdr_reference()
A primitive that provides pointer chasing within structures. The parameter *pp* is the address of the pointer; *size* is the *sizeof* the structure that ∗*pp* points to; and *proc* is an XDR procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

Warning: this routine does not understand NULL pointers. Use **xdr_pointer**() instead.

xdr_setpos()
A macro that invokes the set position routine associated with the XDR stream *xdrs*. The parameter *pos* is a position value obtained from **xdr_getpos**(). This routine returns one if the XDR stream could be repositioned, and zero otherwise.

Warning: it is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

xdr_short()
A filter primitive that translates between C short integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdrstdio_create()
This routine initializes the XDR stream object pointed to by *xdrs*. The XDR stream data is written to, or read from, the Standard I/O stream *file*. The parameter *op* determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

Warning: the destroy routine associated with such XDR streams calls fflush(3) on the file stream, but never fclose(3).

xdr_string()
A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than *maxsize*. Note: *sp* is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

xdr_u_char()
A filter primitive that translates between unsigned C characters and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_hyper()
A filter primitive that translates between unsigned ANSI C long long integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_int()
A filter primitive that translates between C unsigned integers
and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_long()
A filter primitive that translates between C unsigned long integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_longlong_t()
A filter primitive that translates between unsigned ANSI C long long integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_short()
A filter primitive that translates between C unsigned short integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_union()
A filter primitive that translates between a discriminated C union and its corresponding external representation. It first translates the discriminant of the union located at *dscmp*. This discriminant is always an enum_t. Next the union located at *unp* is translated. The parameter *choices* is a pointer to an array of **xdr_discrim**() structures. Each structure contains an ordered pair of [*value*, *proc*]. If the union's discriminant is equal to the associated *value*, then the *proc* is called to translate the union. The end of the **xdr_discrim**() structure array is denoted by a routine of value NULL. If the discriminant is not found in the *choices* array, then the *defaultarm* procedure is called (if it is not NULL). Returns one if it succeeds, zero otherwise.

xdr_vector()
A filter primitive that translates between fixed-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *size* is the element count of the array. The parameter *elsize* is the *sizeof* each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

xdr_void()
This routine always returns one. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

xdr_wrapstring()
A primitive that calls **xdr_string**(*xdrs*, *sp*, *MAXUN.UNSIGNED*); where MAXUN.UNSIGNED is the maximum value of an unsigned integer. **xdr_wrapstring**() is handy because the RPC package passes a maximum of two XDR routines as parameters, and **xdr_string**(), one of the most frequently used primitives, requires three. Returns one if it succeeds, zero otherwise.

SEE ALSO

rpc(3)

The following manuals:

eXternal Data Representation Standard: Protocol Specification.

Sun Microsystems, Inc., USC-ISI, *XDR: External Data Representation Standard*, RFC 1014, USC-ISI, 1014.

NAME

yp_all, **yp_bind**, **yp_first**, **yp_get_default_domain**, **yp_master**, **yp_match**, **yp_next**, **yp_order**, **yp_unbind**, **yperr_string**, **ypprot_err** — Interface to the YP subsystem

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <rpc/rpc.h>
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

int
yp_all(const char *indomain, const char *inmap,
    struct ypall_callback *incallback);

int
yp_bind(const char *dom);

int
yp_first(const char *indomain, const char *inmap, char **outkey,
    int *outkeylen, char **outval, int *outvallen);

int
yp_get_default_domain(char **outdomain);

int
yp_master(const char *indomain, const char *inmap, char **outname);

int
yp_match(const char *indomain, const char *inmap, const char *inkey,
    int inkeylen, char **outval, int *outvallen);

int
yp_next(const char *indomain, const char *inmap, const char *inkey,
    int inkeylen, char **outkey, int *outkeylen, char **outval,
    int *outvallen);

int
yp_order(const char *indomain, const char *inmap, int *outorder);

void
yp_unbind(const char *dom);

char *
yperr_string(int incode);

int
ypprot_err(unsigned int incode);
```

DESCRIPTION

The **ypclnt** suite provides an interface to the YP subsystem. For a general description of the YP subsystem, see yp(8).

For all functions, input values begin with `in` and output values begin with `out`.

Any output values of type *char* ** should be the addresses of uninitialized character pointers. These values will be reset to the null pointer (unless the address itself is the null pointer, in which case the error YPERR_BADARGS will be returned). If necessary, memory will be allocated by the YP client routines using **malloc**(), and the result will be stored in the appropriate output value. If the invocation of a YP client routine doesn't return an error, and an output value is not the null pointer, then this memory should be freed by the user when there is no additional need for the data stored there. For `outkey` and `outval`, two extra bytes of memory are allocated for a '`\n`' and '`\0`', which are not reflected in the values of `outkeylen` or `outvallen`.

All occurrences of `indomain` and `inmap` must be non-null, NUL-terminated strings. All input strings which also have a corresponding length parameter cannot be the null pointer unless the corresponding length value is zero. Such strings need not be NUL-terminated.

All YP lookup calls (the functions **yp_all**(), **yp_first**(), **yp_master**(), **yp_match**(), **yp_next**(), **yp_order**()) require a YP domain name and a YP map name. The default domain name may be obtained by calling **yp_get_default_domain**(), and should thus be used before all other YP calls in a client program. The value it places `outdomain` is suitable for use as the `indomain` parameter to all subsequent YP calls.

In order for YP lookup calls to succeed, the client process must be bound to a YP server process. The client process need not explicitly bind to the server, as it happens automatically whenever a lookup occurs. The function **yp_bind**() is provided for a backup strategy, e.g. a local file, when a YP server process is not available. Each binding uses one socket descriptor on the client process, which may be explicitly freed using **yp_unbind**(), which frees all per-process and per-node resources to bind the domain and marks the domain unbound.

If, during a YP lookup, an RPC failure occurs, the domain used in the lookup is automatically marked unbound and the **ypclnt** layer retries the lookup as long as `ypbind`(8) is running and either the client process cannot bind to a server for the domain specified in the lookup, or RPC requests to the YP server process fail. If an error is not RPC-related, one of the YP error codes described below is returned and control given back to the user code.

The **ypclnt** suite provides the following functionality:

yp_match() Provides the value associated with the given key.

yp_first() Provides the first key-value pair from the given map in the named domain.

yp_next() Provides the next key-value pair in the given map. To obtain the second pair, the `inkey` value should be the `outkey` value provided by the initial call to **yp_first**(). In the general case, the next key-value pair may be obtained by using the `outkey` value from the previous call to **yp_next**() as the value for `inkey`.

Of course, the notions of "first" and "next" are particular to the type of YP map being accessed, and thus there is no guarantee of lexical order. The only guarantees provided with **yp_first**() and **yp_next**(), providing that the same map on the same server is polled repeatedly until **yp_next**() returns YPERR_NOMORE, are that all key-value pairs in that map will be accessed exactly once, and if the entire procedure is repeated, the order will be the same.

If the server is heavily loaded or the server fails for some reason, the domain being used may become unbound. If this happens, and the client process re-binds, the retrieval rules will break: some entries may be seen twice, and others not at all. For this reason, the function **yp_all**() provides a better solution for reading all of the entries in a particular map.

yp_all() This function provides a way to transfer an entire map from the server to the client process with a single request. This transfer uses TCP, unlike all other functions in the **ypclnt** suite, which use UDP. The entire transaction occurs in a single RPC request-response. The third argument to this function provides a way to supply the name of a function to process each key-value pair in the map. **yp_all**() returns after the entire transaction is complete, or the `foreach` function decides that it does not want any more key-value pairs. The third argument to **yp_all**() is:

```
struct ypall_callback *incallback {
        int (*foreach)();
        char *data;
};
```

The *char *data* argument is an opaque pointer for use by the callback function. The `foreach` function should return non-zero when it no longer wishes to process key-value pairs, at which time **yp_all**() returns a value of 0, and is called with the following arguments:

```
int foreach (
        int instatus,
        char *inkey,
        int inkeylen,
        char *inval,
        int invallen,
        char *indata
);
```

Where:

instatus Holds one of the return status values described in ⟨`rpcsvc/yp_prot.h`⟩: see **ypprot_err**() below for a function that will translate YP protocol errors into a **ypclnt** layer error code as described in ⟨`rpcsvc/ypclnt.h`⟩.

inkey, inval The key and value arguments are somewhat different here than described above. In this case, the memory pointed to by *inkey* and *inval* is private to **yp_all**(), and is overwritten with each subsequent key-value pair, thus the `foreach` function should do something useful with the contents of that memory during each iteration. If the key-value pairs are not terminated with either '`\n`' or '`\0`' in the map, then they will not be terminated as such when given to the `foreach` function, either.

indata This is the contents of the `incallback->data` element of the callback structure. It is provided as a means to share state between the `foreach` function and the user code. Its use is completely optional: cast it to something useful or simply ignore it.

yp_order() Returns the order number for a map.

yp_master() Returns the hostname for the machine on which the master YP server process for a map is running.

yperr_string()
Returns a pointer to a NUL-terminated error string that does not contain a '`.`' or '`\n`'.

ypprot_err() Converts a YP protocol error code to a **ypclnt** error code suitable for **yperr_string**().

RETURN VALUES

All functions in the **ypclnt** suite which are of type *int* return 0 upon success or one of the following error codes upon failure:

[YPERR_BADARGS]	The passed arguments to the function are invalid.
[YPERR_BADDB]	The YP map that was polled is defective.
[YPERR_DOMAIN]	Client process cannot bind to server on this YP domain.
[YPERR_KEY]	The key passed does not exist.
[YPERR_MAP]	There is no such map in the server's domain.
[YPERR_DOM]	The local YP domain is not set.
[YPERR_NOMORE]	There are no more records in the queried map.
[YPERR_PMAP]	Cannot communicate with portmapper (see rpcbind(8)).
[YPERR_RESRC]	A resource allocation failure occurred.
[YPERR_RPC]	An RPC failure has occurred. The domain has been marked unbound.
[YPERR_VERS]	Client/server version mismatch. If the server is running version 1 of the YP protocol, **yp_all**() functionality does not exist.
[YPERR_BIND]	Cannot communicate with ypbind(8).
[YPERR_YPERR]	An internal server or client error has occurred.
[YPERR_YPSERV]	The client cannot communicate with the YP server process.

SEE ALSO

malloc(3), yp(8), ypbind(8), ypserv(8)

AUTHORS

Theo De Raadt

NAME

zlib − compression/decompression library

SYNOPSIS

[see *zlib.h* for full description]

DESCRIPTION

The *zlib* library is a general purpose data compression library. The code is thread safe. It provides in-memory compression and decompression functions, including integrity checks of the uncompressed data. This version of the library supports only one compression method (deflation) but other algorithms will be added later and will have the same stream interface.

Compression can be done in a single step if the buffers are large enough (for example if an input file is mmap'ed), or can be done by repeated calls of the compression function. In the latter case, the application must provide more input and/or consume the output (providing more output space) before each call.

The library also supports reading and writing files in *gzip*(1) (.gz) format with an interface similar to that of stdio.

The library does not install any signal handler. The decoder checks the consistency of the compressed data, so the library should never crash even in case of corrupted input.

All functions of the compression library are documented in the file *zlib.h*. The distribution source includes examples of use of the library in the files *example.c* and *minigzip.c*.

Changes to this version are documented in the file *ChangeLog* that accompanies the source, and are concerned primarily with bug fixes and portability enhancements.

A Java implementation of *zlib* is available in the Java Development Kit 1.1:

http://www.javasoft.com/products/JDK/1.1/docs/api/Package-java.util.zip.html

A Perl interface to *zlib*, written by Paul Marquess (pmqs@cpan.org), is available at CPAN (Comprehensive Perl Archive Network) sites, including:

http://www.cpan.org/modules/by-module/Compress/

A Python interface to *zlib*, written by A.M. Kuchling (amk@magnet.com), is available in Python 1.5 and later versions:

http://www.python.org/doc/lib/module-zlib.html

A *zlib* binding for *tcl*(1), written by Andreas Kupries (a.kupries@westend.com), is availlable at:

http://www.westend.com/~kupries/doc/trf/man/man.html

An experimental package to read and write files in .zip format, written on top of *zlib* by Gilles Vollant (info@winimage.com), is available at:

http://www.winimage.com/zLibDll/unzip.html and also in the *contrib/minizip* directory of the main *zlib* web site.

SEE ALSO

The *zlib* web site can be found at either of these locations:

http://www.zlib.org
http://www.gzip.org/zlib/

The data format used by the zlib library is described by RFC (Request for Comments) 1950 to 1952 in the files:

http://www.ietf.org/rfc/rfc1950.txt (concerning zlib format)
http://www.ietf.org/rfc/rfc1951.txt (concerning deflate format)
http://www.ietf.org/rfc/rfc1952.txt (concerning gzip format)

These documents are also available in other formats from:

ftp://ftp.uu.net/graphics/png/documents/zlib/zdoc-index.html

Mark Nelson (markn@ieee.org) wrote an article about *zlib* for the Jan. 1997 issue of Dr. Dobb's Journal; a copy of the article is available at:

http://dogma.net/markn/articles/zlibtool/zlibtool.htm

REPORTING PROBLEMS

Before reporting a problem, please check the *zlib* web site to verify that you have the latest version of *zlib*; otherwise, obtain the latest version and see if the problem still exists. Please read the *zlib* FAQ at:

http://www.gzip.org/zlib/zlib_faq.html

before asking for help. Send questions and/or comments to zlib@gzip.org, or (for the Windows DLL version) to Gilles Vollant (info@winimage.com).

AUTHORS

UNIX manual page by R. P. C. Rodgers, U.S. National Library of Medicine (rodgers@nlm.nih.gov).

NAME

zlib — general purpose compression library

SYNOPSIS

```
#include <zlib.h>
```

Basic functions

```
const char *
zlibVersion(void);

int
deflateInit(z_streamp strm, int level);

int
deflate(z_streamp strm, int flush);

int
deflateEnd(z_streamp strm);

int
inflateInit(z_streamp strm);

int
inflate(z_streamp strm, int flush);

int
inflateEnd(z_streamp strm);
```

Advanced functions

```
int
deflateInit2(z_streamp strm, int level, int method, int windowBits,
    int memLevel, int strategy);

int
deflateSetDictionary(z_streamp strm, const Bytef *dictionary,
    uInt dictLength);

int
deflateCopy(z_streamp dest, z_streamp source);

int
deflateReset(z_streamp strm);

int
deflateParams(z_streamp strm, int level, int strategy);

int
inflateInit2(z_streamp strm, int windowBits);

int
inflateSetDictionary(z_streamp strm, const Bytef *dictionary,
    uInt dictLength);

int
inflateSync(z_streamp strm);

int
inflateReset(z_streamp strm);
```

Utility functions

typedef voidp gzFile ;

int
compress(*Bytef *dest*, *uLongf *destLen*, *const Bytef *source*, *uLong sourceLen*);

int
compress2(*Bytef *dest*, *uLongf *destLen*, *const Bytef *source*, *uLong sourceLen*, *int level*);

int
uncompress(*Bytef *dest*, *uLongf *destLen*, *const Bytef *source*, *uLong sourceLen*);

gzFile
gzopen(*const char *path*, *const char *mode*);

gzFile
gzdopen(*int fd*, *const char *mode*);

int
gzsetparams(*gzFile file*, *int level*, *int strategy*);

int
gzread(*gzFile file*, *voidp buf*, *unsigned len*);

int
gzwrite(*gzFile file*, *const voidp buf*, *unsigned len*);

int
gzprintf(*gzFile file*, *const char *format*, ...*);

int
gzputs(*gzFile file*, *const char *s*);

*char **
gzgets(*gzFile file*, *char *buf*, *int len*);

int
gzputc(*gzFile file*, *int c*);

int
gzgetc(*gzFile file*);

int
gzflush(*gzFile file*, *int flush*);

z_off_t
gzseek(*gzFile file*, *z_off_t offset*, *int whence*);

int
gzrewind(*gzFile file*);

z_off_t
gztell(*gzFile file*);

int
gzeof(*gzFile file*);

int
gzclose(*gzFile file*);

*const char **
gzerror(*gzFile file*, *int *errnum*);

Checksum functions

uLong
adler32(*uLong adler*, *const Bytef *buf*, *uInt len*);

uLong
crc32(*uLong crc*, *const Bytef *buf*, *uInt len*);

DESCRIPTION

This manual page describes the **zlib** general purpose compression library, version 1.1.4.

The **zlib** compression library provides in-memory compression and decompression functions, including integrity checks of the uncompressed data. This version of the library supports only one compression method (deflation) but other algorithms will be added later and will have the same stream interface.

Compression can be done in a single step if the buffers are large enough (for example if an input file is mmap'ed), or can be done by repeated calls of the compression function. In the latter case, the application must provide more input and/or consume the output (providing more output space) before each call.

The library also supports reading and writing files in gzip(1) (.gz) format with an interface similar to that of stdio(3).

The library does not install any signal handler. The decoder checks the consistency of the compressed data, so the library should never crash even in case of corrupted input.

The functions within the library are divided into the following sections:

- Basic functions
- Advanced functions
- Utility functions
- Checksum functions

BASIC FUNCTIONS

*const char ** **zlibVersion**(*void*);

The application can compare **zlibVersion**() and ZLIB_VERSION for consistency. If the first character differs, the library code actually used is not compatible with the ⟨zlib.h⟩ header file used by the application. This check is automatically made by **deflateInit**() and **inflateInit**().

int **deflateInit**(*z_streamp strm*, *int level*);

The **deflateInit**() function initializes the internal stream state for compression. The fields *zalloc*, *zfree*, and *opaque* must be initialized before by the caller. If *zalloc* and *zfree* are set to Z_NULL, **deflateInit**() updates them to use default allocation functions.

The compression level must be Z_DEFAULT_COMPRESSION, or between 0 and 9: 1 gives best speed, 9 gives best compression, 0 gives no compression at all (the input data is simply copied a block at a time).

Z_DEFAULT_COMPRESSION requests a default compromise between speed and compression (currently equivalent to level 6).

deflateInit() returns Z_OK if successful, Z_MEM_ERROR if there was not enough memory, Z_STREAM_ERROR if level is not a valid compression level, Z_VERSION_ERROR if the **zlib** library version (zlib_version) is incompatible with the version assumed by the caller (ZLIB_VERSION). *msg* is set to null if there is no error message. **deflateInit**() does not perform any compression: this will be done by **deflate**().

int **deflate**(*z_streamp strm*, *int flush*);

deflate() compresses as much data as possible, and stops when the input buffer becomes empty or the output buffer becomes full. It may introduce some output latency (reading input without producing any output) except when forced to flush.

The detailed semantics are as follows. **deflate**() performs one or both of the following actions:

Compress more input starting at *next_in* and update *next_in* and *avail_in* accordingly. If not all input can be processed (because there is not enough room in the output buffer), *next_in* and *avail_in* are updated and processing will resume at this point for the next call to **deflate**().

Provide more output starting at *next_out* and update *next_out* and *avail_out* accordingly. This action is forced if the parameter *flush* is non-zero. Forcing *flush* frequently degrades the compression ratio, so this parameter should be set only when necessary (in interactive applications). Some output may be provided even if *flush* is not set.

Before the call to **deflate**(), the application should ensure that at least one of the actions is possible, by providing more input and/or consuming more output, and updating *avail_in* or *avail_out* accordingly; *avail_out* should never be zero before the call. The application can consume the compressed output when it wants, for example when the output buffer is full (avail_out == 0), or after each call to **deflate**(). If **deflate**() returns Z_OK and with zero *avail_out*, it must be called again after making room in the output buffer because there might be more output pending.

If the parameter *flush* is set to Z_SYNC_FLUSH, all pending output is flushed to the output buffer and the output is aligned on a byte boundary, so that the decompressor can get all input data available so far. (In particular, *avail_in* is zero after the call if enough output space has been provided before the call.) Flushing may degrade compression for some compression algorithms and so it should be used only when necessary.

If *flush* is set to Z_FULL_FLUSH, all output is flushed as with Z_SYNC_FLUSH, and the compression state is reset so that decompression can restart from this point if previous compressed data has been damaged or if random access is desired. Using Z_FULL_FLUSH too often can seriously degrade the compression.

If **deflate**() returns with avail_out == 0, this function must be called again with the same value of the flush parameter and more output space (updated *avail_out*), until the flush is complete (**deflate**() returns with non-zero *avail_out*).

If the parameter *flush* is set to Z_FINISH, pending input is processed, pending output is flushed and **deflate**() returns with Z_STREAM_END if there was enough output space; if **deflate**() returns with Z_OK, this function must be called again with Z_FINISH and more output space (updated *avail_out* but no more input data, until it returns with Z_STREAM_END or an error. After **deflate**() has returned Z_STREAM_END, the only possible operations on the stream are **deflateReset**() or **deflateEnd**().

Z_FINISH can be used immediately after **deflateInit**() if all the compression is to be done in a single step. In this case, *avail_out* must be at least 0.1% larger than *avail_in* plus 12 bytes. If **deflate**() does not return Z_STREAM_END, then it must be called again as described above.

deflate() sets strm->adler to the Adler-32 checksum of all input read so far (that is, *total_in* bytes).

deflate() may update *data_type* if it can make a good guess about the input data type (Z_ASCII or Z_BINARY). If in doubt, the data is considered binary. This field is only for information purposes and does not affect the compression algorithm in any manner.

deflate() returns Z_OK if some progress has been made (more input processed or more output produced), Z_STREAM_END if all input has been consumed and all output has been produced (only when *flush* is set to Z_FINISH), Z_STREAM_ERROR if the stream state was inconsistent (for example, if *next_in* or *next_out* was NULL), Z_BUF_ERROR if no progress is possible (for example, *avail_in* or *avail_out* was zero).

int **deflateEnd**(*z_streamp strm*);

All dynamically allocated data structures for this stream are freed. This function discards any unprocessed input and does not flush any pending output.

deflateEnd() returns Z_OK if successful, Z_STREAM_ERROR if the stream state was inconsistent, Z_DATA_ERROR if the stream was freed prematurely (some input or output was discarded). In the error case, *msg* may be set but then points to a static string (which must not be deallocated).

int **inflateInit**(*z_streamp strm*);

The **inflateInit**() function initializes the internal stream state for decompression. The fields *next_in*, *avail_in*, *zalloc*, *zfree*, and *opaque* must be initialized before by the caller. If *next_in* is not Z_NULL and *avail_in* is large enough (the exact value depends on the compression method), **inflateInit**() determines the compression method from the **zlib** header and allocates all data structures accordingly; otherwise the allocation will be deferred to the first call to **inflate**(). If *zalloc* and *zfree* are set to Z_NULL, **inflateInit**() updates them to use default allocation functions.

inflateInit() returns Z_OK if successful, Z_MEM_ERROR if there was not enough memory, Z_VERSION_ERROR if the **zlib** library version is incompatible with the version assumed by the caller. *msg* is set to null if there is no error message. **inflateInit**() does not perform any decompression apart from reading the **zlib** header if present: this will be done by **inflate**(). (So *next_in* and *avail_in* may be modified, but *next_out* and *avail_out* are unchanged.)

int **inflate**(*z_streamp strm*, *int flush*);

inflate() decompresses as much data as possible, and stops when the input buffer becomes empty or the output buffer becomes full. It may introduce some output latency (reading input without producing any output) except when forced to flush.

The detailed semantics are as follows. **inflate**() performs one or both of the following actions:

Decompress more input starting at *next_in* and update *next_in* and *avail_in* accordingly. If not all input can be processed (because there is not enough room in the output buffer), *next_in* is updated and processing will resume at this point for the next call to **inflate**().

Provide more output starting at *next_out* and update *next_out* and *avail_out* accordingly. **inflate**() provides as much output as possible, until there is no more input data or no more space in the output buffer (see below about the flush parameter).

Before the call to **inflate**(), the application should ensure that at least one of the actions is possible, by providing more input and/or consuming more output, and updating the next_* and avail_* values accordingly. The application can consume the uncompressed output when it wants, for example when the output buffer is full (avail_out == 0), or after each call to **inflate**(). If **inflate**() returns Z_OK and with zero *avail_out*, it must be called again after making room in

the output buffer because there might be more output pending.

If the parameter *flush* is set to Z_SYNC_FLUSH, **inflate**() flushes as much output as possible to the output buffer. The flushing behavior of **inflate**() is not specified for values of the flush parameter other than Z_SYNC_FLUSH and Z_FINISH, but the current implementation actually flushes as much output as possible anyway.

inflate() should normally be called until it returns Z_STREAM_END or an error. However if all decompression is to be performed in a single step (a single call to inflate), the parameter *flush* should be set to Z_FINISH. In this case all pending input is processed and all pending output is flushed; *avail_out* must be large enough to hold all the uncompressed data. (The size of the uncompressed data may have been saved by the compressor for this purpose.) The next operation on this stream must be **inflateEnd**() to deallocate the decompression state. The use of Z_FINISH is never required, but can be used to inform **inflate**() that a faster routine may be used for the single **inflate**() call.

If a preset dictionary is needed at this point (see **inflateSetDictionary**() below), **inflate**() sets strm->adler to the Adler-32 checksum of the dictionary chosen by the compressor and returns Z_NEED_DICT; otherwise it sets strm->adler to the Adler-32 checksum of all output produced so far (that is, *total_out* bytes) and returns Z_OK, Z_STREAM_END, or an error code as described below. At the end of the stream, **inflate**() checks that its computed Adler-32 checksum is equal to that saved by the compressor and returns Z_STREAM_END only if the checksum is correct.

inflate() returns Z_OK if some progress has been made (more input processed or more output produced), Z_STREAM_END if the end of the compressed data has been reached and all uncompressed output has been produced, Z_NEED_DICT if a preset dictionary is needed at this point, Z_DATA_ERROR if the input data was corrupted (input stream not conforming to the **zlib** format or incorrect Adler-32 checksum), Z_STREAM_ERROR if the stream structure was inconsistent (for example, if *next_in* or *next_out* was NULL), Z_MEM_ERROR if there was not enough memory, Z_BUF_ERROR if no progress is possible or if there was not enough room in the output buffer when Z_FINISH is used. In the Z_DATA_ERROR case, the application may then call **inflateSync**() to look for a good compression block.

int **inflateEnd**(*z_streamp strm*);

All dynamically allocated data structures for this stream are freed. This function discards any unprocessed input and does not flush any pending output.

inflateEnd() returns Z_OK if successful, or Z_STREAM_ERROR if the stream state was inconsistent. In the error case, *msg* may be set but then points to a static string (which must not be deallocated).

ADVANCED FUNCTIONS

The following functions are needed only in some special applications.

int **deflateInit2**(*z_streamp strm*, *int level*, *int method*, *int windowBits*, *int memLevel*, *int strategy*);

This is another version of **deflateInit**() with more compression options. The fields *next_in*, *zalloc*, *zfree*, and *opaque* must be initialized before by the caller.

The *method* parameter is the compression method. It must be Z_DEFLATED in this version of the library.

The *windowBits* parameter is the base two logarithm of the window size (the size of the history buffer). It should be in the range 8..15 for this version of the library. Larger values of this parameter result in better compression at the expense of memory usage. The default value is 15 if

deflateInit() is used instead.

The *memLevel* parameter specifies how much memory should be allocated for the internal compression state. memLevel=1 uses minimum memory but is slow and reduces compression ratio; memLevel=9 uses maximum memory for optimal speed. The default value is 8. See ⟨zconf.h⟩ for total memory usage as a function of *windowBits* and *memLevel*.

The *strategy* parameter is used to tune the compression algorithm. Use the value Z_DEFAULT_STRATEGY for normal data; Z_FILTERED for data produced by a filter (or predictor); or Z_HUFFMAN_ONLY to force Huffman encoding only (no string match). Filtered data consists mostly of small values with a somewhat random distribution. In this case, the compression algorithm is tuned to compress them better. The effect of Z_FILTERED is to force more Huffman coding and less string matching; it is somewhat intermediate between Z_DEFAULT and Z_HUFFMAN_ONLY. The *strategy* parameter only affects the compression ratio but not the correctness of the compressed output, even if it is not set appropriately.

deflateInit2() returns Z_OK if successful, Z_MEM_ERROR if there was not enough memory, Z_STREAM_ERROR if a parameter is invalid (such as an invalid method). *msg* is set to null if there is no error message. **deflateInit2**() does not perform any compression: this will be done by **deflate**().

int **deflateSetDictionary**(*z_streamp strm*, *const Bytef *dictionary*, *uInt dictLength*);

Initializes the compression dictionary from the given byte sequence without producing any compressed output. This function must be called immediately after **deflateInit**(), **deflateInit2**(), or **deflateReset**(), before any call to **deflate**(). The compressor and decompressor must use exactly the same dictionary (see **inflateSetDictionary**()).

The dictionary should consist of strings (byte sequences) that are likely to be encountered later in the data to be compressed, with the most commonly used strings preferably put towards the end of the dictionary. Using a dictionary is most useful when the data to be compressed is short and can be predicted with good accuracy; the data can then be compressed better than with the default empty dictionary.

Depending on the size of the compression data structures selected by **deflateInit**() or **deflateInit2**(), a part of the dictionary may in effect be discarded, for example if the dictionary is larger than the window size in **deflate**() or **deflate2**(). Thus the strings most likely to be useful should be put at the end of the dictionary, not at the front.

Upon return of this function, strm->adler is set to the Adler-32 value of the dictionary; the decompressor may later use this value to determine which dictionary has been used by the compressor. (The Adler-32 value applies to the whole dictionary even if only a subset of the dictionary is actually used by the compressor.)

deflateSetDictionary() returns Z_OK if successful, or Z_STREAM_ERROR if a parameter is invalid (such as NULL dictionary) or the stream state is inconsistent (for example if **deflate**() has already been called for this stream or if the compression method is bsort). **deflateSetDictionary**() does not perform any compression: this will be done by **deflate**().

int **deflateCopy**(*z_streamp dest*, *z_streamp source*);

The **deflateCopy**() function sets the destination stream as a complete copy of the source stream.

This function can be useful when several compression strategies will be tried, for example when there are several ways of pre-processing the input data with a filter. The streams that will be discarded should then be freed by calling **deflateEnd**(). Note that **deflateCopy**() duplicates the

internal compression state which can be quite large, so this strategy is slow and can consume lots of memory.

deflateCopy() returns Z_OK if successful, Z_MEM_ERROR if there was not enough memory, Z_STREAM_ERROR if the source stream state was inconsistent (such as *zalloc* being NULL). *msg* is left unchanged in both source and destination.

int **deflateReset**(*z_streamp strm*);

This function is equivalent to **deflateEnd**() followed by **deflateInit**(), but does not free and reallocate all the internal compression state. The stream will keep the same compression level and any other attributes that may have been set by **deflateInit2**().

deflateReset() returns Z_OK if successful, or Z_STREAM_ERROR if the source stream state was inconsistent (such as *zalloc* or *state* being NULL).

int **deflateParams**(*z_streamp strm*, *int level*, *int strategy*);

The **deflateParams**() function dynamically updates the compression level and compression strategy. The interpretation of level and strategy is as in **deflateInit2**(). This can be used to switch between compression and straight copy of the input data, or to switch to a different kind of input data requiring a different strategy. If the compression level is changed, the input available so far is compressed with the old level (and may be flushed); the new level will take effect only at the next call to **deflate**().

Before the call to **deflateParams**(), the stream state must be set as for a call to **deflate**(), since the currently available input may have to be compressed and flushed. In particular, strm->avail_out must be non-zero.

deflateParams() returns Z_OK if successful, Z_STREAM_ERROR if the source stream state was inconsistent or if a parameter was invalid, or Z_BUF_ERROR if strm->avail_out was zero.

int **inflateInit2**(*z_streamp strm*, *int windowBits*);

This is another version of **inflateInit**() with an extra parameter. The fields *next_in*, *avail_in*, *zalloc*, *zfree*, and *opaque* must be initialized before by the caller.

The *windowBits* parameter is the base two logarithm of the maximum window size (the size of the history buffer). It should be in the range 8..15 for this version of the library. The default value is 15 if **inflateInit**() is used instead. If a compressed stream with a larger window size is given as input, **inflate**() will return with the error code Z_DATA_ERROR instead of trying to allocate a larger window.

inflateInit2() returns Z_OK if successful, Z_MEM_ERROR if there was not enough memory, Z_STREAM_ERROR if a parameter is invalid (such as a negative *memLevel*). *msg* is set to null if there is no error message. **inflateInit2**() does not perform any decompression apart from reading the **zlib** header if present: this will be done by **inflate**(). (So *next_in* and *avail_in* may be modified, but *next_out* and *avail_out* are unchanged.)

int **inflateSetDictionary**(*z_streamp strm*, *const Bytef *dictionary*, *uInt dictLength*);

Initializes the decompression dictionary from the given uncompressed byte sequence. This function must be called immediately after a call to **inflate**() if this call returned Z_NEED_DICT. The dictionary chosen by the compressor can be determined from the Adler-32 value returned by this call to **inflate**(). The compressor and decompressor must use exactly the same dictionary (see **deflateSetDictionary**()).

inflateSetDictionary() returns Z_OK if successful, Z_STREAM_ERROR if a parameter is invalid (such as NULL dictionary) or the stream state is inconsistent, Z_DATA_ERROR if the given dictionary doesn't match the expected one (incorrect Adler-32 value). **inflateSetDictionary**() does not perform any decompression: this will be done by subsequent calls of **inflate**().

int **inflateSync**(*z_streamp strm*);

Skips invalid compressed data until a full flush point (see above the description of **deflate**() with Z_FULL_FLUSH) can be found, or until all available input is skipped. No output is provided.

inflateSync() returns Z_OK if a full flush point has been found, Z_BUF_ERROR if no more input was provided, Z_DATA_ERROR if no flush point has been found, or Z_STREAM_ERROR if the stream structure was inconsistent. In the success case, the application may save the current value of *total_in* which indicates where valid compressed data was found. In the error case, the application may repeatedly call **inflateSync**(), providing more input each time, until success or end of the input data.

int **inflateReset**(*z_streamp strm*);

This function is equivalent to **inflateEnd**() followed by **inflateInit**(), but does not free and reallocate all the internal decompression state. The stream will keep attributes that may have been set by **inflateInit2**().

inflateReset() returns Z_OK if successful, or Z_STREAM_ERROR if the source stream state was inconsistent (such as *zalloc* or *state* being NULL).

UTILITY FUNCTIONS

The following utility functions are implemented on top of the basic stream-oriented functions. To simplify the interface, some default options are assumed (compression level and memory usage, standard memory allocation functions). The source code of these utility functions can easily be modified if you need special options.

int **compress**(*Bytef *dest*, *uLongf *destLen*, *const Bytef *source*, *uLong sourceLen*);

The **compress**() function compresses the source buffer into the destination buffer. *sourceLen* is the byte length of the source buffer. Upon entry, *destLen* is the total size of the destination buffer, which must be at least 0.1% larger than *sourceLen* plus 12 bytes. Upon exit, *destLen* is the actual size of the compressed buffer. This function can be used to compress a whole file at once if the input file is mmap'ed.

compress() returns Z_OK if successful, Z_MEM_ERROR if there was not enough memory, or Z_BUF_ERROR if there was not enough room in the output buffer.

int **compress2**(*Bytef *dest*, *uLongf *destLen*, *const Bytef *source*, *uLong sourceLen*, *int level*);

The **compress2**() function compresses the source buffer into the destination buffer. The *level* parameter has the same meaning as in **deflateInit**(). *sourceLen* is the byte length of the source buffer. Upon entry, *destLen* is the total size of the destination buffer, which must be at least 0.1% larger than *sourceLen* plus 12 bytes. Upon exit, *destLen* is the actual size of the compressed buffer.

compress2() returns Z_OK if successful, Z_MEM_ERROR if there was not enough memory, Z_BUF_ERROR if there was not enough room in the output buffer, or Z_STREAM_ERROR if the level parameter is invalid.

int **uncompress**(*Bytef *dest*, *uLongf *destLen*, *const Bytef *source*, *uLong sourceLen*);

The **uncompress**() function decompresses the source buffer into the destination buffer. *sourceLen* is the byte length of the source buffer. Upon entry, *destLen* is the total size of the destination buffer, which must be large enough to hold the entire uncompressed data. (The size of the uncompressed data must have been saved previously by the compressor and transmitted to the decompressor by some mechanism outside the scope of this compression library.) Upon exit, *destLen* is the actual size of the compressed buffer. This function can be used to decompress a whole file at once if the input file is mmap'ed.

uncompress() returns Z_OK if successful, Z_MEM_ERROR if there was not enough memory, Z_BUF_ERROR if there was not enough room in the output buffer, or Z_DATA_ERROR if the input data was corrupted.

gzFile **gzopen**(*const char *path*, *const char *mode*);

The **gzopen**() function opens a gzip (.gz) file for reading or writing. The mode parameter is as in fopen(3) ("rb" or "wb") but can also include a compression level ("wb9") or a strategy: 'f' for filtered data, as in "wb6f"; 'h' for Huffman only compression, as in "wb1h". (See the description of **deflateInit2**() for more information about the strategy parameter.)

gzopen() can be used to read a file which is not in gzip format; in this case **gzread**() will directly read from the file without decompression.

gzopen() returns NULL if the file could not be opened or if there was insufficient memory to allocate the (de)compression state; errno can be checked to distinguish the two cases (if errno is zero, the **zlib** error is Z_MEM_ERROR).

gzFile **gzdopen**(*int fd*, *const char *mode*);

The **gzdopen**() function associates a gzFile with the file descriptor *fd*. File descriptors are obtained from calls like open(2), dup(2), creat(3), pipe(2), or fileno(3) (if the file has been previously opened with fopen(3)). The *mode* parameter is as in **gzopen**().

The next call to **gzclose**() on the returned gzFile will also close the file descriptor fd, just like fclose(fdopen(fd), mode) closes the file descriptor fd. If you want to keep fd open, use gzdopen(dup(fd), mode).

gzdopen() returns NULL if there was insufficient memory to allocate the (de)compression state.

int **gzsetparams**(*gzFile file*, *int level*, *int strategy*);

The **gzsetparams**() function dynamically updates the compression level or strategy. See the description of **deflateInit2**() for the meaning of these parameters.

gzsetparams() returns Z_OK if successful, or Z_STREAM_ERROR if the file was not opened for writing.

int **gzread**(*gzFile file*, *voidp buf*, *unsigned len*);

The **gzread**() function reads the given number of uncompressed bytes from the compressed file. If the input file was not in gzip format, **gzread**() copies the given number of bytes into the buffer.

gzread() returns the number of uncompressed bytes actually read (0 for end of file, −1 for error).

int **gzwrite**(*gzFile file*, *const voidp buf*, *unsigned len*);

The **gzwrite**() function writes the given number of uncompressed bytes into the compressed file. **gzwrite**() returns the number of uncompressed bytes actually written (0 in case of error).

int **gzprintf**(*gzFile file*, *const char *format*, ...);

The **gzprintf**() function converts, formats, and writes the args to the compressed file under control of the format string, as in fprintf(3). **gzprintf**() returns the number of uncompressed bytes actually written (0 in case of error).

int **gzputs**(*gzFile file*, *const char *s*);

The **gzputs**() function writes the given null-terminated string to the compressed file, excluding the terminating null character.

gzputs() returns the number of characters written, or −1 in case of error.

*char * * **gzgets**(*gzFile file*, *char *buf*, *int len*);

The **gzgets**() function reads bytes from the compressed file until len−1 characters are read, or a newline character is read and transferred to *buf*, or an end-of-file condition is encountered. The string is then terminated with a null character.

gzgets() returns *buf*, or Z_NULL in case of error.

int **gzputc**(*gzFile file*, *int c*);

The **gzputc**() function writes *c*, converted to an unsigned char, into the compressed file. **gzputc**() returns the value that was written, or −1 in case of error.

int **gzgetc**(*gzFile file*);

The **gzgetc**() function reads one byte from the compressed file. **gzgetc**() returns this byte or −1 in case of end of file or error.

int **gzflush**(*gzFile file*, *int flush*);

The **gzflush**() function flushes all pending output into the compressed file. The parameter *flush* is as in the **deflate**() function. The return value is the **zlib** error number (see function **gzerror**() below). **gzflush**() returns Z_OK if the flush parameter is Z_FINISH and all output could be flushed.

gzflush() should be called only when strictly necessary because it can degrade compression.

z_off_t **gzseek**(*gzFile file*, *z_off_t offset*, *int whence*);

Sets the starting position for the next **gzread**() or **gzwrite**() on the given compressed file. The offset represents a number of bytes in the uncompressed data stream. The whence parameter is defined as in lseek(2); the value SEEK_END is not supported.

If the file is opened for reading, this function is emulated but can be extremely slow. If the file is opened for writing, only forward seeks are supported; **gzseek**() then compresses a sequence of zeroes up to the new starting position.

gzseek() returns the resulting offset location as measured in bytes from the beginning of the uncompressed stream, or −1 in case of error, in particular if the file is opened for writing and the new starting position would be before the current position.

int **gzrewind**(*gzFile file*);

The **gzrewind**() function rewinds the given *file*. This function is supported only for reading.

gzrewind(file) is equivalent to (int)gzseek(file, 0L, SEEK_SET).

z_off_t **gztell**(*gzFile file*);

The **gztell**() function returns the starting position for the next **gzread**() or **gzwrite**() on the given compressed file. This position represents a number of bytes in the uncompressed data stream.

gztell(file) is equivalent to gzseek(file, 0L, SEEK_CUR).

int **gzeof**(*gzFile file*);

The **gzeof**() function returns 1 when EOF has previously been detected reading the given input stream, otherwise zero.

int **gzclose**(*gzFile file*);

The **gzclose**() function flushes all pending output if necessary, closes the compressed file and deallocates all the (de)compression state. The return value is the **zlib** error number (see function **gzerror**() below).

const char * **gzerror**(*gzFile file*, *int *errnum*);

The **gzerror**() function returns the error message for the last error which occurred on the given compressed *file*. *errnum* is set to the **zlib** error number. If an error occurred in the file system and not in the compression library, *errnum* is set to Z_ERRNO and the application may consult errno to get the exact error code.

CHECKSUM FUNCTIONS

These functions are not related to compression but are exported anyway because they might be useful in applications using the compression library.

uLong **adler32**(*uLong adler*, *const Bytef *buf*, *uInt len*);

The **adler32**() function updates a running Adler-32 checksum with the bytes buf[0..len-1] and returns the updated checksum. If *buf* is NULL, this function returns the required initial value for the checksum.

An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much faster. Usage example:

```
uLong adler = adler32(0L, Z_NULL, 0);

while (read_buffer(buffer, length) != EOF) {
adler = adler32(adler, buffer, length);
}
if (adler != original_adler) error();
```

uLong **crc32**(*uLong crc*, *const Bytef *buf*, *uInt len*);

The **crc32**() function updates a running CRC with the bytes buf[0..len-1] and returns the updated CRC. If *buf* is NULL, this function returns the required initial value for the CRC. Pre- and post-conditioning (one's complement) is performed within this function so it shouldn't be done by the application. Usage example:

```
uLong crc = crc32(0L, Z_NULL, 0);

while (read_buffer(buffer, length) != EOF) {
crc = crc32(crc, buffer, length);
}
if (crc != original_crc) error();
```

STRUCTURES

```
struct internal_state;

typedef struct z_stream_s {
    Bytef    *next_in;  /* next input byte */
    uInt     avail_in;  /* number of bytes available at next_in */
    uLong    total_in;  /* total nb of input bytes read so far */

    Bytef    *next_out; /* next output byte should be put there */
    uInt     avail_out; /* remaining free space at next_out */
    uLong    total_out; /* total nb of bytes output so far */

    char     *msg;      /* last error message, NULL if no error */
    struct internal_state FAR *state; /* not visible by applications */

    alloc_func zalloc;  /* used to allocate the internal state */
    free_func  zfree;   /* used to free the internal state */
    voidpf     opaque;  /* private data object passed to zalloc and zfree*/

    int     data_type;  /*best guess about the data type: ascii or binary*/
    uLong   adler;      /* Adler-32 value of the uncompressed data */
    uLong   reserved;   /* reserved for future use */
} z_stream;

typedef z_stream FAR * z_streamp;
```

The application must update *next_in* and *avail_in* when *avail_in* has dropped to zero. It must update *next_out* and *avail_out* when *avail_out* has dropped to zero. The application must initialize *zalloc*, *zfree*, and *opaque* before calling the init function. All other fields are set by the compression library and must not be updated by the application.

The *opaque* value provided by the application will be passed as the first parameter for calls to **zalloc**() and **zfree**(). This can be useful for custom memory management. The compression library attaches no meaning to the *opaque* value.

zalloc must return Z_NULL if there is not enough memory for the object. If **zlib** is used in a multithreaded application, *zalloc* and *zfree* must be thread safe.

On 16-bit systems, the functions *zalloc* and *zfree* must be able to allocate exactly 65536 bytes, but will not be required to allocate more than this if the symbol MAXSEG_64K is defined (see ⟨zconf.h⟩).

WARNING: On MSDOS, pointers returned by *zalloc* for objects of exactly 65536 bytes *must* have their offset normalized to zero. The default allocation function provided by this library ensures this (see zutil.c). To reduce memory requirements and avoid any allocation of 64K objects, at the expense of compression ratio, compile the library with -DMAX_WBITS=14 (see ⟨zconf.h⟩).

The fields *total_in* and *total_out* can be used for statistics or progress reports. After compression, *total_in* holds the total size of the uncompressed data and may be saved for use in the decompressor (particularly if the decompressor wants to decompress everything in a single step).

CONSTANTS

```
#define Z_NO_FLUSH      0
#define Z_PARTIAL_FLUSH 1 /* will be removed, use Z_SYNC_FLUSH instead */
#define Z_SYNC_FLUSH    2
#define Z_FULL_FLUSH    3
```

```
#define Z_FINISH        4
/* Allowed flush values; see deflate() below for details */

#define Z_OK            0
#define Z_STREAM_END    1
#define Z_NEED_DICT     2
#define Z_ERRNO        (-1)
#define Z_STREAM_ERROR (-2)
#define Z_DATA_ERROR   (-3)
#define Z_MEM_ERROR    (-4)
#define Z_BUF_ERROR    (-5)
#define Z_VERSION_ERROR (-6)
/* Return codes for the compression/decompression functions.
 * Negative values are errors,
 * positive values are used for special but normal events.
 */

#define Z_NO_COMPRESSION         0
#define Z_BEST_SPEED             1
#define Z_BEST_COMPRESSION       9
#define Z_DEFAULT_COMPRESSION  (-1)
/* compression levels */

#define Z_FILTERED            1
#define Z_HUFFMAN_ONLY        2
#define Z_DEFAULT_STRATEGY    0
/* compression strategy; see deflateInit2() below for details */

#define Z_BINARY   0
#define Z_ASCII    1
#define Z_UNKNOWN  2
/* Possible values of the data_type field */

#define Z_DEFLATED   8
/* The deflate compression method
 * (the only one supported in this version)
*/

#define Z_NULL  0  /* for initializing zalloc, zfree, opaque */

#define zlib_version zlibVersion()
/* for compatibility with versions < 1.0.2 */
```

VARIOUS HACKS

deflateInit and inflateInit are macros to allow checking the **zlib** version and the compiler's view of *z_stream*.

int **deflateInit**_(*z_stream strm*, *int level*, *const char *version*, *int stream_size*);

int **inflateInit**_(*z_stream strm*, *const char *version*, *int stream_size*);

int **deflateInit2_**(*z_stream strm*, *int level*, *int method*, *int windowBits*, *int memLevel*, *int strategy*, *const char *version*, *int stream_size*)

int **inflateInit2_**(*z_stream strm*, *int windowBits*, *const char *version*, *int stream_size*);

const char * **zError**(*int err*);

int **inflateSyncPoint**(*z_streamp z*);

const uLongf * **get_crc_table**(*void*);

SEE ALSO

RFC 1950 ZLIB Compressed Data Format Specification.
RFC 1951 DEFLATE Compressed Data Format Specification.
RFC 1952 GZIP File Format Specification.

`http://www.gzip.org/zlib/`

HISTORY

This manual page is based on an HTML version of ⟨`zlib.h`⟩ converted by piaip ⟨piaip@csie.ntu.edu.tw⟩ and was converted to mdoc format by the OpenBSD project.

AUTHORS

Jean-loup Gailly ⟨jloup@gzip.org⟩
Mark Adler ⟨madler@alumni.caltech.edu⟩

NAME

zopen — compressed stream open function

SYNOPSIS

```
#include <stdio.h>

FILE *
zopen(const char *path, const char *mode, int bits);
```

DESCRIPTION

The **zopen**() function opens the compressed file whose name is the string pointed to by *path* and associates a stream with it.

The argument *mode* points to one of the following one-character strings:

"r" Open compressed file for reading. The stream is positioned at the beginning of the file.

"w" Truncate file to zero length or create compressed file for writing. The stream is positioned at the beginning of the file.

Any created files will have mode "S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH" (0666), as modified by the process' umask value (see umask(2)).

Files may only be read or written. Seek operations are not allowed.

The *bits* argument, if non-zero, is set to the bits code limit. If zero, the default is 16. See **compress**(*1*) for more information.

RETURN VALUES

Upon successful completion **zopen**() returns a FILE pointer. Otherwise, NULL is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] The *mode* or *bits* arguments specified to **zopen**() were invalid.

[EFTYPE] The compressed file starts with an invalid header, or the compressed file is compressed with more bits than can be handled.

The **zopen**() function may also fail and set *errno* for any of the errors specified for the routines fopen(3) or funopen(3).

SEE ALSO

compress(1), fopen(3), funopen(3)

HISTORY

The **zopen** function first appeared in 4.4BSD.

BUGS

The **zopen**() function may not be portable to systems other than BSD.