NAME

intro — introduction to kernel internals

DESCRIPTION

This section contains information related to the internal operation of the system kernel. It describes function interfaces and variables of use to the systems and device driver programmer.

In addition to the normal man page format, the kernel pages include an additional section:

CODE REFERENCES

Contains the pathname(s) of the source file(s) which contain the definition and/or source code of the variables or functions being documented.

MEMORY MANAGEMENT

Machine-dependent swap interface. See cpu_swapout(9).

Introduction to kernel memory allocators. See memoryallocators(9).

Machine-dependent portion of the virtual memory system. See pmap(9).

Virtual memory system external interface. See uvm(9).

I/O SUBSYSTEM

Buffer cache interfaces. See buffercache(9).

Device buffer queues. See bufq(9).

Initiate I/O on raw devices. See physio(9).

I/O descriptor allocation interface. See getiobuf(9).

PROCESS CONTROL

Machine-dependent process exit. See cpu_exit(9).

Idle CPU while waiting for work. See cpu_idle(9).

Finish a fork operation. See cpu_lwp_fork(9).

Switch to another light weight process. See ctxsw(9).

Current process and processor. See curproc(9).

Set process uid and gid. See do_setresuid(9).

New processes and kernel threads. See fork1(9), kthread(9).

Context switch notification. See cpu_need_resched(9).

Process scheduling subsystem. See scheduler(9).

Software signal facilities. See signal(9).

Suspend the scheduler. See suspendsched(9).

Return path to user-mode execution. See userret(9).

FILE SYSTEM

High-level file operations. See dofileread(9).

Convert an extended attribute namespace identifier to a string and vice versa. See extattr(9).

Operations on file entries. See `file`(9).

In-kernel, file-system independent, file-meta data association. See `fileassoc`(9).

File descriptor tables and operations. See `filedesc`(9).

File descriptor owner handling functions. See `fsetown`(9).

File system suspension helper subsystem. See `fstrans`(9).

Pathname lookup, cache and management. See `namei`(9), `namecache`(9), `pathname`(9).

Kernel interface to file systems. See `vfs`(9).

Kernel representation of a file or directory and vnode attributes. See `vnode`(9), `vattr`(9).

NETWORKING

Kernel interfaces for manipulating output queues on network interfaces. See `altq`(9).

Externally visible ARP functions. See `arp`(9).

Ethernet and FDDI driver support functions and macros. See `ethersubr`(9).

Core 802.11 network stack functions and rate adaptation based on received signal strength. See `ieee80211`(9), `rssadapt`(9).

Compute Internet checksum. See `in_cksum`(9).

Look up the IPv4 source address best matching an IPv4 destination. See `in_getifa`(9).

Functions and macros for managing memory used by networking code. See `mbuf`(9).

Packet filter interface. See `pfil`(9).

Route callout functions. See `rt_timer`(9).

TCP congestion control API. See `tcp_congctl`(9).

LOCKING AND INTERRUPT CONTROL

Condition variables. See `condvar`(9).

Kernel lock functions. See `lock`(9).

Memory barriers. See `mb`(9).

Mutual exclusion primitives. See `mutex`(9).

Restartable atomic sequences. See `ras`(9).

Reader / writer lock primitives. See `rwlock`(9).

Machine-independent software interrupt framework. See `softintr`(9).

Functions to modify system interrupt priority level. See `spl`(9).

Functions ro raise the system priority level. See `splraiseipl`(9).

SECURITY

Kernel authorization framework. See `kauth`(9).

API for cryptographic services in the kernel. See `opencrypto`(9).

Security model development guidelines. See `secmodel`(9).

SYSTEM TIME CONTROL

Execute a function after a specified length of time. See `callout`(9).

Microsecond delay. See `delay`(9).

Real-time timer. See `hardclock`(9).

System clock frequency. See `hz`(9).

Initialization of system time and time-of-day clock support. See `inittodr`(9), `todr`(9).

Check that a timeval value is valid, and correct. See `itimerfix`(9).

System time variables. See `timecounter`(9).

Realtime system clock. See `microtime`(9).

Get the time elapsed since boot. See `microuptime`(9).

Convert milliseconds to system clock ticks. See `mstohz`(9).

Function to help implement rate-limited actions. See `ppsratecheck`(9).

Function to help implement rate-limited actions. See `ratecheck`(9).

Set battery-backed clock from system time. See `resettodr`(9).

System time variables. See `time_second`(9).

KERNEL AND USER SPACE DATA COPY FUNCTIONS

Kernel space to/from user space copy functions. See `copy`(9).

Store data to user-space. See `store`(9).

Fetch data from user-space. See `fetch`(9).

Move data described by a struct uio. See `uiomove`(9).

MACHINE DEPENDENT KERNEL FUNCTIONS

Machine-dependent clock setup interface. See `cpu_initclocks`(9).

Machine-dependent process core dump interface. See `cpu_coredump`(9).

Machine-dependent kernel core dumps. See `cpu_dumpconf`(9).

Unique CPU identification number See `cpu_number`(9).

Halt or reboot the system See `cpu_reboot`(9).

Machine-dependent root file system setup See `cpu_rootconf`(9).

Machine-dependent CPU startup See `cpu_startup`(9).

Disk label management routines. See `disklabel`(9).

DEVICE CONFIGURATION

Autoconfiguration frame-work. See `autoconf`(9).

Description of a device driver. See `driver`(9).

The autoconfiguration framework ''device definition'' language. See `config`(9).

Machine-dependent device autoconfiguration. See `cpu_configure`(9).

MI DEVICE DRIVER API

Bus and Machine Independent DMA Mapping Interface. See `bus_dma`(9).

Bus space manipulation functions. See `bus_space`(9).

Generic disk framework. See `disk`(9).

Hardware-assisted data mover interface. See `dmover`(9). Generic event counter framework. See `evcnt`(9).

Firmware loader API for device drivers. See `firmload`(9).

How to implement a new ioctl call to access device drivers. See `ioctl`(9).

Extensible line discipline framework. See `linedisc`(9).

CONSOLE DEVICES

Console magic key sequence management. See `cnmagic`(9).

Console access interface. See `cons`(9).

Raster display operations. See `rasops`(9).

Generic virtual console framework. See `vcons`(9).

Machine-independent console support. See `wscons`(9).

DEVICE SPECIFIC IMPLEMENTATION

Interface between low and high level audio drivers. See `audio`(9).

Bluetooth Device/Protocol API. See `bluetooth`(9).

Support for CardBus PC-Card devices. See `cardbus`(9).

VESA Display Data Channel V2. See `ddc`(9).

VESA Extended Display Identification Data. See `edid`(9).

Inter IC (I2C) bus. See `iic`(9).

Baseboard I/O control ASIC for DEC TURBOchannel systems. See `ioasic`(9).

Industry-standard Architecture. See `isa`(9).

Introduction to ISA Plug-and-Play support. See `isapnp`(9).

MicroChannel Architecture bus. See `mca`(9).

PPBUS microseqencer developer's guide. See `microseq`(9).

Peripheral Component Interconnect. See `pci`(9).

Perform PCI bus configuration. See `pci_configure_bus`(9).

PCI bus interrupt manipulation functions. See `pci_intr`(9).

PC keyboard port interface. See `pckbport`(9).

Support for PCMCIA PC-Card devices. See `pcmcia`(9).

User-space interface to ppbus parallel port. See `ppi`(9).

Interface between low and high level radio drivers. See `radio`(9).

Functions to make a device available for entropy collection. See rnd(9).

SCSI/ATAPI middle-layer interface. See scsipi(9).

TURBOchannel bus. See tc(9).

USB tty support. See ucom(9).

USB device drivers interface. See usbdi(9).

Versa Module Euroboard bus. See vme(9).

Machine-independent IDE/ATAPI driver. See wdc(9).

KERNEL EVENT

Functions to add or remove kernel event filters. See kfilter_register(9).

Functions to raise kernel event. See knote(9).

Record and wakeup select requests. See selrecord(9).

Simple do-it-in-thread-context framework. See workqueue(9).

KERNEL HELPER FUNCTIONS

Kernel expression verification macros. See KASSERT(9).

Convert a single byte between (unsigned) packed bcd and binary. See bcdtobin(9).

Bitmask output conversion. See bitmask_snprintf(9).

General purpose extent manager. See extent(9).

Compare integers. See imax(9).

Kernel formatted output conversion. See kprintf(9).

Data comparing, moving, copying, setting and cleaning. See memcmp(9), memmove(9), memcpy(9), memset(9), bcmp(9), bcopy(9), bzero(9), kcopy(9).

Log a message from the kernel through the /dev/klog device. See log(9).

Bring down system on fatal error. See panic(9).

MISC

Run all power hooks. See dopowerhooks(9).

Run all shutdown hooks. See doshutdownhooks(9).

Kernel internal error numbers. See errno(9).

Kernel hash functions, hash table construction and destruction. See hash(9), hashinit(9).

Format a number into a human readable form. See humanize_number(9).

Machine-dependent interface to ipkdb. See ipkdb(9).

Options string management. See optstr(9).

Performs pattern matching on strings. See pmatch(9).

Hardware Performance Monitoring Interface. See pmc(9).

Add or remove a power change hook. See powerhook_establish(9).

Add or remove a shutdown hook. See `shutdownhook_establish`(9).

Non-local jumps. See `setjmp`(9).

System variable control interfaces. See `sysctl`(9).

HISTORY

The NetBSD kernel internals section first appeared in NetBSD 1.2.

NAME

KASSERT, **KDASSERT** — kernel expression verification macros

SYNOPSIS

void
KASSERT(*expression*);

void
KDASSERT(*expression*);

DESCRIPTION

These machine independent assertion-checking macros cause a kernel `panic`(9) if the given *expression* evaluates to false.

KASSERT() tests are included only in kernels compiled with the `DIAGNOSTIC` configuration option. In a kernel that does not have this configuration option, the **KASSERT**() macro is defined to be a no-op.

KDASSERT() tests are included only in kernels compiled with the `DEBUG` configuration option. **KDASSERT**() and **KASSERT**() are identical except for the controlling option (`DEBUG` vs `DIAGNOSTIC`).

The panic message will display the style of assertion (debugging vs. diagnostic), the expression that failed and the filename, and line number the failure happened on.

SEE ALSO

`config`(1), `panic`(9), `printf`(9)

AUTHORS

These macros were written by Chris G. Demetriou ⟨cgd@netbsd.org⟩.

NAME

LWP_CACHE_CREDS — synchronize LWP credential with process credential

SYNOPSIS

```
#include <sys/lwp.h>

void
LWP_CACHE_CREDS(lwp_t *l, struct proc *p);
```

DESCRIPTION

LWP_CACHE_CREDS() updates the LWP's cached credential to match with the process' credential if the latter has been changed after the last synchronization.

Each LWPs have its cached credential so that it can be used without worrying about potential of other LWP changing the process' credential. **kauth_cred_get**() returns the cached credential.

LWP_CACHE_CREDS() is called by MD entry code for system call and various traps. LWPs which can live in kernel for long period should call **LWP_CACHE_CREDS**() by itsself to refresh its credential.

LWP_CACHE_CREDS() takes the following arguments.

l The calling lwp.

p The process which the lwp *l* belongs to.

LWP_CACHE_CREDS() might be implemented as a macro.

SEE ALSO

intro(9), kauth(9)

NAME

RUN_ONCE — Run a function exactly once

SYNOPSIS

```
#include <sys/once.h>

ONCE_DECL(control);

int
RUN_ONCE(once_t *control, int (*init_func)(void));
```

DESCRIPTION

RUN_ONCE() provides a functionality similar to **pthread_once**(*3*). It ensures that, for a given *control*, **init_func**() is executed (successfully) exactly once. It is considered as a successful execution if and only if **init_func**() returned 0. As long as there was no successful execution, **RUN_ONCE**() will try again each time it is called.

RUN_ONCE() can sleep if it's called concurrently.

RETURN VALUES

On failure, **RUN_ONCE**() returns what **init_func**() returned. Otherwise, it returns 0.

EXAMPLES

The following example shows how **RUN_ONCE**() is used. Regardless of how many times **some_func**() is executed, **init_func**() will be executed exactly once.

```
static int
init_func(void)
{

        /*
         * do some initialization.
         */

        return 0; /* success */
}

int
some_func(void)
{
        static DECL_ONCE(control);

        RUN_ONCE(&control, init_func);

        /*
         * we are sure that init_func has already been completed here.
         */
}
```

SEE ALSO

pthread_once(3), condvar(9), intro(9)

NAME

ALTQ — kernel interfaces for manipulating output queues on network interfaces

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>

void
IFQ_ENQUEUE(struct ifaltq *ifq, struct mbuf *m, struct altq_pktattr *pattr,
    int err);

void
IFQ_DEQUEUE(struct ifaltq *ifq, struct mbuf *m);

void
IFQ_POLL(struct ifaltq *ifq, struct mbuf *m);

void
IFQ_PURGE(struct ifaltq *ifq);

void
IFQ_CLASSIFY(struct ifaltq *ifq, struct mbuf *m, int af,
    struct altq_pktattr *pattr);

void
IFQ_IS_EMPTY(struct ifaltq *ifq);

void
IFQ_SET_MAXLEN(struct ifaltq *ifq, int len);

void
IFQ_INC_LEN(struct ifaltq *ifq);

void
IFQ_DEC_LEN(struct ifaltq *ifq);

void
IFQ_INC_DROPS(struct ifaltq *ifq);

void
IFQ_SET_READY(struct ifaltq *ifq);
```

DESCRIPTION

The **ALTQ** system is a framework to manage queueing disciplines on network interfaces. **ALTQ** introduces new macros to manipulate output queues. The output queue macros are used to abstract queue operations and not to touch the internal fields of the output queue structure. The macros are independent from the **ALTQ** implementation, and compatible with the traditional `ifqueue` macros for ease of transition.

IFQ_ENQUEUE() enqueues a packet *m* to the queue *ifq*. The underlying queueing discipline may discard the packet. *err* is set to 0 on success, or `ENOBUFS` if the packet is discarded. *m* will be freed by the device driver on success or by the queueing discipline on failure, so that the caller should not touch *m* after calling **IFQ_ENQUEUE**().

IFQ_DEQUEUE() dequeues a packet from the queue. The dequeued packet is returned in *m*, or *m* is set to `NULL` if no packet is dequeued. The caller must always check *m* since a non-empty queue could return `NULL` under rate-limiting.

IFQ_POLL() returns the next packet without removing it from the queue. It is guaranteed by the underlying queueing discipline that **IFQ_DEQUEUE**() immediately after **IFQ_POLL**() returns the same packet.

IFQ_PURGE() discards all the packets in the queue. The purge operation is needed since a non-work conserving queue cannot be emptied by a dequeue loop.

IFQ_CLASSIFY() classifies a packet to a scheduling class, and returns the result in *pattr*.

IFQ_IS_EMPTY() can be used to check if the queue is empty. Note that **IFQ_DEQUEUE**() could still return NULL if the queueing discipline is non-work conserving.

IFQ_SET_MAXLEN() sets the queue length limit to the default FIFO queue.

IFQ_INC_LEN() and **IFQ_DEC_LEN**() increment or decrement the current queue length in packets.

IFQ_INC_DROPS() increments the drop counter and is equal to **IF_DROP**(). It is defined for naming consistency.

IFQ_SET_READY() sets a flag to indicate this driver is converted to use the new macros. **ALTQ** can be enabled only on interfaces with this flag.

COMPATIBILITY

ifaltq structure

In order to keep compatibility with the existing code, the new output queue structure ifaltq has the same fields. The traditional **IF_XXX**() macros and the code directly referencing the fields within if_snd still work with ifaltq. (Once we finish conversions of all the drivers, we no longer need these fields.)

```
            ##old-style##                          |            ##new-style##
                                                   |
 struct ifqueue {                                  | struct ifaltq {
    struct mbuf *ifq_head;                         |    struct mbuf *ifq_head;
    struct mbuf *ifq_tail;                         |    struct mbuf *ifq_tail;
    int          ifq_len;                          |    int          ifq_len;
    int          ifq_maxlen;                       |    int          ifq_maxlen;
    int          ifq_drops;                        |    int          ifq_drops;
 };                                                |    /* altq related fields */
                                                   |    ......
                                                   | };
                                                   |
```

The new structure replaces struct ifqueue in struct ifnet.

```
            ##old-style##                          |            ##new-style##
                                                   |
 struct ifnet {                                    | struct ifnet {
     ....                                          |     ....
                                                   |
     struct ifqueue if_snd;                        |     struct ifaltq if_snd;
                                                   |
     ....                                          |     ....
 };                                                | };
                                                   |
```

The (simplified) new **IFQ_XXX**() macros looks like:

```
#ifdef ALTQ
#define IFQ_DEQUEUE(ifq, m)                  \
        if (ALTQ_IS_ENABLED((ifq))           \
                ALTQ_DEQUEUE((ifq), (m));    \
```

```
                else                                    \
                        IF_DEQUEUE((ifq), (m));
        #else
        #define IFQ_DEQUEUE(ifq, m)   IF_DEQUEUE((ifq), (m));
        #endif
```

Enqueue operation

The semantics of the enqueue operation are changed. In the new style, enqueue and packet drop are combined since they cannot be easily separated in many queueing disciplines. The new enqueue operation corresponds to the following macro that is written with the old macros.

```
#define IFQ_ENQUEUE(ifq, m, pattr, err)                  \
do {                                                      \
        if (ALTQ_IS_ENABLED((ifq)))                       \
                ALTQ_ENQUEUE((ifq), (m), (pattr), (err)); \
        else {                                            \
                if (IF_QFULL((ifq))) {                    \
                        m_freem((m));                     \
                        (err) = ENOBUFS;                  \
                } else {                                  \
                        IF_ENQUEUE((ifq), (m));           \
                        (err) = 0;                        \
                }                                         \
        }                                                 \
        if ((err))                                        \
                (ifq)->ifq_drops++;                       \
} while (/*CONSTCOND*/ 0)
```

IFQ_ENQUEUE() does the following:

- queue a packet
- drop (and free) a packet if the enqueue operation fails

If the enqueue operation fails, *err* is set to ENOBUFS. *m* is freed by the queueing discipline. The caller should not touch mbuf after calling **IFQ_ENQUEUE**() so that the caller may need to copy *m_pkthdr.len* or *m_flags* field beforehand for statistics. The caller should not use **senderr**() since mbuf was already freed.

The new style **if_output**() looks as follows:

```
            ##old-style##                                ##new-style##
                                                    |
 int                                                | int
 ether_output(ifp, m0, dst, rt0)                    | ether_output(ifp, m0, dst, rt0)
 {                                                  | {
     ......                                         |     ......
                                                    |
                                                    |     mflags = m->m_flags;
                                                    |     len = m->m_pkthdr.len;
     s = splimp();                                  |     s = splimp();
     if (IF_QFULL(&ifp->if_snd)) {                  |     IFQ_ENQUEUE(&ifp->if_snd, m,
                                                    |                 NULL, error);
         IF_DROP(&ifp->if_snd);                     |     if (error != 0) {
         splx(s);                                   |         splx(s);
         senderr(ENOBUFS);                          |         return (error);
     }                                              |     }
```

```
        IF_ENQUEUE(&ifp->if_snd, m);      |
        ifp->if_obytes +=                 |     ifp->if_obytes += len;
                        m->m_pkthdr.len;  |
        if (m->m_flags & M_MCAST)         |     if (mflags & M_MCAST)
            ifp->if_omcasts++;            |         ifp->if_omcasts++;
                                          |
        if ((ifp->if_flags & IFF_OACTIVE) |     if ((ifp->if_flags & IFF_OACTIVE)
            == 0)                         |         == 0)
            (*ifp->if_start)(ifp);        |         (*ifp->if_start)(ifp);
        splx(s);                          |     splx(s);
        return (error);                   |     return (error);
                                          |
    bad:                                  | bad:
        if (m)                            |     if (m)
            m_freem(m);                   |         m_freem(m);
        return (error);                   |     return (error);
    }                                     | }
                                          |
```

Classifier

The classifier mechanism is currently implemented in **if_output**(). `struct altq_pktattr` is used to store the classifier result, and it is passed to the enqueue function. (We will change the method to tag the classifier result to mbuf in the future.)

```
int
ether_output(ifp, m0, dst, rt0)
{
        ......
        struct altq_pktattr pktattr;

        ......

        /* classify the packet before prepending link-headers */
        IFQ_CLASSIFY(&ifp->if_snd, m, dst->sa_family, &pktattr);

        /* prepend link-level headers */
        ......

        IFQ_ENQUEUE(&ifp->if_snd, m, &pktattr, error);

        ......
}
```

HOW TO CONVERT THE EXISTING DRIVERS

First, make sure the corresponding **if_output**() is already converted to the new style.

Look for *if_snd* in the driver. You will probably need to make changes to the lines that include *if_snd*.

Empty check operation

If the code checks *ifq_head* to see whether the queue is empty or not, use **IFQ_IS_EMPTY**().

```
            ##old-style##                          ##new-style##
                                          |
```

```
if (ifp->if_snd.ifq_head != NULL)     | if (IFQ_IS_EMPTY(&ifp->if_snd) == 0)
                                      |
```

Note that **IFQ_POLL**() can be used for the same purpose, but **IFQ_POLL**() could be costly for a complex scheduling algorithm since **IFQ_POLL**() needs to run the scheduling algorithm to select the next packet. On the other hand, **IFQ_IS_EMPTY**() checks only if there is any packet stored in the queue. Another difference is that even when **IFQ_IS_EMPTY**() is `false`, **IFQ_DEQUEUE**() could still return `NULL` if the queue is under rate-limiting.

Dequeue operation

Replace **IF_DEQUEUE**() by **IFQ_DEQUEUE**(). Always check whether the dequeued mbuf is `NULL` or not. Note that even when **IFQ_IS_EMPTY**() is `false`, **IFQ_DEQUEUE**() could return `NULL` due to rate-limiting.

```
            ##old-style##                     ##new-style##
                                      |
IF_DEQUEUE(&ifp->if_snd, m);          | IFQ_DEQUEUE(&ifp->if_snd, m);
                                      | if (m == NULL)
                                      |     return;
                                      |
```

A driver is supposed to call **if_start**() from transmission complete interrupts in order to trigger the next dequeue.

Poll-and-dequeue operation

If the code polls the packet at the head of the queue and actually uses the packet before dequeueing it, use **IFQ_POLL**() and **IFQ_DEQUEUE**().

```
            ##old-style##                     ##new-style##
                                      |
m = ifp->if_snd.ifq_head;             | IFQ_POLL(&ifp->if_snd, m);
if (m != NULL) {                      | if (m != NULL) {
                                      |
    /* use m to get resources */      |     /* use m to get resources */
    if (something goes wrong)         |     if (something goes wrong)
        return;                       |         return;
                                      |
    IF_DEQUEUE(&ifp->if_snd, m);      |     IFQ_DEQUEUE(&ifp->if_snd, m);
                                      |
    /* kick the hardware */           |     /* kick the hardware */
}                                     | }
                                      |
```

It is guaranteed that **IFQ_DEQUEUE**() immediately after **IFQ_POLL**() returns the same packet. Note that they need to be guarded by **splimp**() if called from outside of **if_start**().

Eliminating IF_PREPEND

If the code uses **IF_PREPEND**(), you have to eliminate it since the prepend operation is not possible for many queueing disciplines. A common use of **IF_PREPEND**() is to cancel the previous dequeue operation. You have to convert the logic into poll-and-dequeue.

```
            ##old-style##                     ##new-style##
                                      |
IF_DEQUEUE(&ifp->if_snd, m);          | IFQ_POLL(&ifp->if_snd, m);
if (m != NULL) {                      | if (m != NULL) {
                                      |
```

```
    if (something_goes_wrong) {          |     if (something_goes_wrong) {
        IF_PREPEND(&ifp->if_snd, m);     |
        return;                          |         return;
    }                                    |     }
                                         |
                                         |     /* at this point, the driver
                                         |      * is committed to send this
                                         |      * packet.
                                         |      */
                                         |     IFQ_DEQUEUE(&ifp->if_snd, m);
                                         |
    /* kick the hardware */              |     /* kick the hardware */
}                                        | }
                                         |
```

Purge operation

Use **IFQ_PURGE**() to empty the queue. Note that a non-work conserving queue cannot be emptied by a dequeue loop.

```
            ##old-style##                            ##new-style##
                                         |
while (ifp->if_snd.ifq_head != NULL) {   |  IFQ_PURGE(&ifp->if_snd);
    IF_DEQUEUE(&ifp->if_snd, m);         |
    m_freem(m);                          |
}                                        |
                                         |
```

Attach routine

Use **IFQ_SET_MAXLEN**() to set *ifq_maxlen* to *len*. Add **IFQ_SET_READY**() to show this driver is converted to the new style. (This is used to distinguish new-style drivers.)

```
            ##old-style##                            ##new-style##
                                         |
ifp->if_snd.ifq_maxlen = qsize;          | IFQ_SET_MAXLEN(&ifp->if_snd, qsize);
                                         | IFQ_SET_READY(&ifp->if_snd);
if_attach(ifp);                          | if_attach(ifp);
                                         |
```

Other issues

The new macros for statistics:

```
            ##old-style##                            ##new-style##
                                         |
IF_DROP(&ifp->if_snd);                   | IFQ_INC_DROPS(&ifp->if_snd);
                                         |
ifp->if_snd.ifq_len++;                   | IFQ_INC_LEN(&ifp->if_snd);
                                         |
ifp->if_snd.ifq_len--;                   | IFQ_DEC_LEN(&ifp->if_snd);
                                         |
```

Some drivers instruct the hardware to invoke transmission complete interrupts only when it thinks necessary. Rate-limiting breaks its assumption.

How to convert drivers using multiple ifqueues

Some (pseudo) devices (such as slip) have another `ifqueue` to prioritize packets. It is possible to eliminate the second queue since **ALTQ** provides more flexible mechanisms but the following shows how to keep the original behavior.

```
struct sl_softc {
        struct  ifnet sc_if;            /* network-visible interface */
        ...
        struct  ifqueue sc_fastq;       /* interactive output queue */
        ...
};
```

The driver doesn't compile in the new model since it has the following line (*if_snd* is no longer a type of `struct ifqueue`).

```
        struct ifqueue *ifq = &ifp->if_snd;
```

A simple way is to use the original **IF_XXX**() macros for *sc_fastq* and use the new **IFQ_XXX**() macros for *if_snd*. The enqueue operation looks like:

```
            ##old-style##                           ##new-style##
                                                |
 struct ifqueue *ifq = &ifp->if_snd;            | struct ifqueue *ifq = NULL;
                                                |
 if (ip->ip_tos & IPTOS_LOWDELAY)               | if ((ip->ip_tos & IPTOS_LOWDELAY) &&
     ifq = &sc->sc_fastq;                       | !ALTQ_IS_ENABLED(&sc->sc_if.if_snd)) {
                                                |     ifq = &sc->sc_fastq;
 if (IF_QFULL(ifq)) {                           |     if (IF_QFULL(ifq)) {
     IF_DROP(ifq);                              |         IF_DROP(ifq);
     m_freem(m);                                |         m_freem(m);
     splx(s);                                   |         error = ENOBUFS;
     sc->sc_if.if_oerrors++;                    |     } else {
     return (ENOBUFS);                          |         IF_ENQUEUE(ifq, m);
 }                                              |         error = 0;
 IF_ENQUEUE(ifq, m);                            |     }
                                                | } else
                                                |     IFQ_ENQUEUE(&sc->sc_if.if_snd,
                                                |                 m, NULL, error);
                                                |
                                                | if (error) {
                                                |     splx(s);
                                                |     sc->sc_if.if_oerrors++;
                                                |     return (error);
                                                | }
 if ((sc->sc_oqlen =                            | if ((sc->sc_oqlen =
      sc->sc_ttyp->t_outq.c_cc) == 0)           |      sc->sc_ttyp->t_outq.c_cc) == 0)
     slstart(sc->sc_ttyp);                      |     slstart(sc->sc_ttyp);
 splx(s);                                       | splx(s);
                                                |
```

The dequeue operations looks like:

```
            ##old-style##                           ##new-style##
                                                |
 s = splimp();                                  | s = splimp();
 IF_DEQUEUE(&sc->sc_fastq, m);                  | IF_DEQUEUE(&sc->sc_fastq, m);
 if (m == NULL)                                 | if (m == NULL)
```

```
    IF_DEQUEUE(&sc->sc_if.if_snd, m); |     IFQ_DEQUEUE(&sc->sc_if.if_snd, m);
splx(s);                              | splx(s);
                                      |
```

QUEUEING DISCIPLINES

Queueing disciplines need to maintain *ifq_len* (used by **IFQ_IS_EMPTY**()). Queueing disciplines also need to guarantee the same mbuf is returned if **IFQ_DEQUEUE**() is called immediately after **IFQ_POLL**().

SEE ALSO

pf(4), altq.conf(5), pf.conf(5), altqd(8), tbrconfig(8)

HISTORY

The **ALTQ** system first appeared in March 1997.

NAME

arc4random — arc4 random number generator

SYNOPSIS

```
#include <sys/types.h>
#include <sys/systm.h>

uint32_t
arc4random(void);
```

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 such as **random**().

SEE ALSO

arc4random(3), rnd(4)

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

arp, **arp_ifinit**, **arpresolve**, **arpintr** — externally visible ARP functions

SYNOPSIS

```
#include <netinet/if_inarp.h>

void
arp_ifinit(struct ifnet *ifp, struct ifaddr *ifa);

int
arpresolve(struct ifnet *ifp, struct rtentry *rt, struct mbuf *m,
    struct sockaddr *dst, u_char *desten);

void
arpintr();
```

DESCRIPTION

The **arp** functions provide the interface between the **arp** module and the network drivers which need **arp** functionality. Such drivers must request the *arp* attribute in their "files" declaration.

arp_ifinit() Sets up the **arp** specific fields in *ifa*. Additionally, it sends out a gratuitous **arp** request on *ifp*, so that other machines are warned that we have a (new) address and duplicate addresses can be detected.

You must call this in your drivers' ioctl function when you get a SIOCSIFADDR request with an AF_INET address family.

arpresolve() is called by network output functions to resolve an IPv4 address. If no *rt* is given, a new one is looked up or created. If the passed or found *rt* does not contain a valid gateway link level address, a pointer to the packet in *m* is stored in the route entry, possibly replacing older stored packets, and an **arp** request is sent instead. When an **arp** reply is received, the last held packet is send. Otherwise, the looked up address is returned and written into the storage *desten* points to. **arpresolve**() returns 1, if a valid address was stored to *desten*, and the packet can be sent immediately. Else a 0 is returned.

arpintr() When an **arp** packet is received, the network driver (class) input interrupt handler queues the packet on the arpintrq queue, and requests an **arpintr**() soft interrupt callback. **arpintr**() dequeues the packets, performs sanity checks and calls (for IPv4 **arp** packets, which are the only ones supported currently) the **in_arpinput**() function. **in_arpinput**() either generates a reply to request packets, and adds the sender address translation to the routing table, if a matching route entry is found. If the route entry contained a pointer to a held packet, that packet is sent.

SEE ALSO

ether_ifattach(9)
Plummer, D., "RFC826", An Ethernet Address Resolution Protocol.

AUTHORS

UCB CSRG (original implementation)
Ignatios Souvatzis (support for non-Ethernet)

CODE REFERENCES

The ARP code is implemented in sys/net/if_arp.h, sys/netinet/if_inarp.h and sys/netinet/if_arp.c.

STANDARDS

RFC 826

HISTORY

Rewritten to support other than Ethernet link level addresses in NetBSD 1.3.

NAME

audio — interface between low and high level audio drivers

DESCRIPTION

The audio device driver is divided into a high level, hardware independent layer, and a low level hardware dependent layer. The interface between these is the *audio_hw_if* structure.

```
struct audio_hw_if {
        int     (*open)(void *, int);
        void    (*close)(void *);
        int     (*drain)(void *);

        int     (*query_encoding)(void *, struct audio_encoding *);
        int     (*set_params)(void *, int, int,
                    audio_params_t *, audio_params_t *,
                    stream_filter_list_t *, stream_filter_list_t *);
        int     (*round_blocksize)(void *, int, int, const audio_params_t *);

        int     (*commit_settings)(void *);

        int     (*init_output)(void *, void *, int);
        int     (*init_input)(void *, void *, int);
        int     (*start_output)(void *, void *, int, void (*)(void *),
                     void *);
        int     (*start_input)(void *, void *, int, void (*)(void *),
                    void *);
        int     (*halt_output)(void *);
        int     (*halt_input)(void *);

        int     (*speaker_ctl)(void *, int);
#define SPKR_ON  1
#define SPKR_OFF 0

        int     (*getdev)(void *, struct audio_device *);
        int     (*setfd)(void *, int);

        int     (*set_port)(void *, mixer_ctrl_t *);
        int     (*get_port)(void *, mixer_ctrl_t *);

        int     (*query_devinfo)(void *, mixer_devinfo_t *);

        void    *(*allocm)(void *, int, size_t, struct malloc_type *, int);
        void    (*freem)(void *, void *, struct malloc_type *);
        size_t  (*round_buffersize)(void *, int, size_t);
        paddr_t (*mappage)(void *, void *, off_t, int);

        int     (*get_props)(void *);

        int     (*trigger_output)(void *, void *, void *, int,
                    void (*)(void *), void *, const audio_params_t *);
        int     (*trigger_input)(void *, void *, void *, int,
                    void (*)(void *), void *, const audio_params_t *);
```

```
        int     (*dev_ioctl)(void *, u_long, void *, int, struct lwp *);
        int     (*powerstate)(void *, int);
#define AUDIOPOWER_ON  1
#define AUDIOPOWER_OFF 0
};

typedef struct audio_params {
        u_int   sample_rate;    /* sample rate */
        u_int   encoding;       /* e.g. mu-law, linear, etc */
        u_int   precision;      /* bits/subframe */
        u_int   validbits;      /* valid bits in a subframe */
        u_int   channels;       /* mono(1), stereo(2) */
} audio_params_t;
```

The high level audio driver attaches to the low level driver when the latter calls *audio_attach_mi*. This call should be

```
    void
    audio_attach_mi(ahwp, hdl, dev)
        struct audio_hw_if *ahwp;
        void *hdl;
        struct device *dev;
```

The *audio_hw_if* struct is as shown above. The *hdl* argument is a handle to some low level data structure. It is sent as the first argument to all the functions in *audio_hw_if* when the high level driver calls them. *dev* is the device struct for the hardware device.

The upper layer of the audio driver allocates one buffer for playing and one for recording. It handles the buffering of data from the user processes in these. The data is presented to the lower level in smaller chunks, called blocks. If, during playback, there is no data available from the user process when the hardware request another block a block of silence will be used instead. Furthermore, if the user process does not read data quickly enough during recording data will be thrown away.

The fields of *audio_hw_if* are described in some more detail below. Some fields are optional and can be set to 0 if not needed.

`int open(void *hdl, int flags)`
: optional, is called when the audio device is opened. It should initialize the hardware for I/O. Every successful call to *open* is matched by a call to *close*. Return 0 on success, otherwise an error code.

`void close(void *hdl)`
: optional, is called when the audio device is closed.

`int drain(void *hdl)`
: optional, is called before the device is closed or when `AUDIO_DRAIN` is called. It should make sure that no samples remain in to be played that could be lost when *close* is called. Return 0 on success, otherwise an error code.

`int query_encoding(void *hdl, struct audio_encoding *ae)`
: is used when `AUDIO_GETENC` is called. It should fill the *audio_encoding* structure and return 0 or, if there is no encoding with the given number, return EINVAL.

```
int set_params(void *hdl, int setmode, int usemode,
        audio_params_t *play, audio_params_t *rec,
```

```
stream_filter_list_t *pfil, stream_filter_list_t *rfil)
```

Called to set the audio encoding mode. *setmode* is a combination of the AUMODE_RECORD and AUMODE_PLAY flags to indicate which mode(s) are to be set. *usemode* is also a combination of these flags, but indicates the current mode of the device (i.e., the value of *mode* in the *audio_info* struct).

The *play* and *rec* structures contain the encoding parameters that should be set. The values of the structures may also be modified if the hardware cannot be set to exactly the requested mode (e.g., if the requested sampling rate is not supported, but one close enough is).

If the hardware requires software assistance with some encoding (e.g., it might be lacking mu-law support) it should fill the *pfil* for playing or *rfil* for recording with conversion information. For example, if *play* requests [8000Hz, mu-law, 8/8bit, 1ch] and the hardware does not support 8bit mu-law, but 16bit slinear_le, the driver should call `pfil->append()` with *pfil*, *mulaw_to_slinear16*, and audio_params_t representing [8000Hz, slinear_le, 16/16bit, 2ch]. If the driver needs multiple conversions, a conversion nearest to the hardware should be set to the head of *pfil* or *rfil*. The definition of `stream_filter_list_t` follows:

```
typedef struct stream_filter_list {
        void (*append)(struct stream_filter_list *,
                       stream_filter_factory_t,
                       const audio_params_t *);
        void (*prepend)(struct stream_filter_list *,
                        stream_filter_factory_t,
                        const audio_params_t *);
        void (*set)(struct stream_filter_list *, int,
                    stream_filter_factory_t,
                    const audio_params_t *);
        int req_size;
        struct stream_filter_req {
                stream_filter_factory_t *factory;
                audio_params_t param; /* from-param for recording,
                                         to-param for playing */
        } filters[AUDIO_MAX_FILTERS];
} stream_filter_list_t;
```

For playing, *pfil* constructs conversions as follows:

```
        (play) == write(2) input
          |     pfil->filters[pfil->req_size-1].factory
        (pfil->filters[pfil->req_size-1].param)
          |     pfil->filters[pfil->req_size-2].factory
          :
          |     pfil->filters[1].factory
        (pfil->filters[1].param)
          |     pfil->filters[0].factory
        (pfil->filters[0].param)  == hardware input
```

For recording, *rfil* constructs conversions as follows:

```
        (rfil->filters[0].param) == hardware output
          |     rfil->filters[0].factory
        (rfil->filters[1].param)
          |     rfil->filters[1].factory
          :
```

```
  |     rfil->filters[rfil->req_size-2].factory
(rfil->filters[rfil->req_size-1].param)
  |     rfil->filters[rfil->req_size-1].factory
(rec)  == read(2) output
```

If the device does not have the AUDIO_PROP_INDEPENDENT property the same value is passed in both *play* and *rec* and the encoding parameters from *play* is copied into *rec* after the call to *set_params*. Return 0 on success, otherwise an error code.

`int round_blocksize(void *hdl, int bs, int mode, const audio_params_t *param)`

optional, is called with the block size, *bs*, that has been computed by the upper layer, *mode*, AUMODE_PLAY or AUMODE_RECORD, and *param*, encoding parameters for the hardware. It should return a block size, possibly changed according to the needs of the hardware driver.

`int commit_settings(void *hdl)`

optional, is called after all calls to *set_params*, and *set_port*, are done. A hardware driver that needs to get the hardware in and out of command mode for each change can save all the changes during previous calls and do them all here. Return 0 on success, otherwise an error code.

`int init_output(void *hdl, void *buffer, int size)`

optional, is called before any output starts, but when the total *size* of the output *buffer* has been determined. It can be used to initialize looping DMA for hardware that needs that. Return 0 on success, otherwise an error code.

`int init_input(void *hdl, void *buffer, int size)`

optional, is called before any input starts, but when the total *size* of the input *buffer* has been determined. It can be used to initialize looping DMA for hardware that needs that. Return 0 on success, otherwise an error code.

`int start_output(void *hdl, void *block, int blksize, void (*intr)(void*), void *intrarg)`

is called to start the transfer of *blksize* bytes from *block* to the audio hardware. The call should return when the data transfer has been initiated (normally with DMA). When the hardware is ready to accept more samples the function *intr* should be called with the argument *intrarg*. Calling *intr* will normally initiate another call to *start_output*. Return 0 on success, otherwise an error code.

`int start_input(void *hdl, void *block, int blksize, void (*intr)(void*), void *intrarg)`

is called to start the transfer of *blksize* bytes to *block* from the audio hardware. The call should return when the data transfer has been initiated (normally with DMA). When the hardware is ready to deliver more samples the function *intr* should be called with the argument *intrarg*. Calling *intr* will normally initiate another call to *start_input*. Return 0 on success, otherwise an error code.

`int halt_output(void *hdl)`

is called to abort the output transfer (started by *start_output*) in progress. Return 0 on success, otherwise an error code.

`int halt_input(void *hdl)`

is called to abort the input transfer (started by *start_input*) in progress. Return 0 on success, otherwise an error code.

`int speaker_ctl(void *hdl, int on)`
: optional, is called when a half duplex device changes between playing and recording. It can, e.g., be used to turn on and off the speaker. Return 0 on success, otherwise an error code.

`int getdev(void *hdl, struct audio_device *ret)`
: Should fill the *audio_device* struct with relevant information about the driver. Return 0 on success, otherwise an error code.

`int setfd(void *hdl, int fd)`
: optional, is called when `AUDIO_SETFD` is used, but only if the device has AUDIO_PROP_FULLDUPLEX set. Return 0 on success, otherwise an error code.

`int set_port(void *hdl, mixer_ctrl_t *mc)`
: is called in when `AUDIO_MIXER_WRITE` is used. It should take data from the *mixer_ctrl_t* struct at set the corresponding mixer values. Return 0 on success, otherwise an error code.

`int get_port(void *hdl, mixer_ctrl_t *mc)`
: is called in when `AUDIO_MIXER_READ` is used. It should fill the *mixer_ctrl_t* struct. Return 0 on success, otherwise an error code.

`int query_devinfo(void *hdl, mixer_devinfo_t *di)`
: is called in when `AUDIO_MIXER_DEVINFO` is used. It should fill the *mixer_devinfo_t* struct. Return 0 on success, otherwise an error code.

`void *allocm(void *hdl, int direction, size_t size, struct malloc_type *type, int flags)`

: optional, is called to allocate the device buffers. If not present `malloc`(9) is used instead (with the same arguments but the first two). The reason for using a device dependent routine instead of `malloc`(9) is that some buses need special allocation to do DMA. Returns the address of the buffer, or 0 on failure.

`void freem(void *hdl, void *addr, struct malloc_type *type)`
: optional, is called to free memory allocated by *alloc*. If not supplied `free`(9) is used.

`size_t round_buffersize(void *hdl, int direction, size_t bufsize)`
: optional, is called at startup to determine the audio buffer size. The upper layer supplies the suggested size in *bufsize*, which the hardware driver can then change if needed. E.g., DMA on the ISA bus cannot exceed 65536 bytes.

`paddr_t mappage(void *hdl, void *addr, off_t offs, int prot)`

: optional, is called for `mmap`(2). Should return the map value for the page at offset *offs* from address *addr* mapped with protection *prot*. Returns -1 on failure, or a machine dependent opaque value on success.

`int get_props(void *hdl)`
: Should return the device properties; i.e., a combination of AUDIO_PROP_xxx.

`int trigger_output(void *hdl, void *start, void *end,`
`int blksize, void (*intr)(void*), void *intrarg,`

`const audio_params_t *param)`

: optional, is called to start the transfer of data from the circular buffer delimited by *start* and *end* to the audio hardware, parameterized as in *param*. The call should return when the data transfer has been initiated (normally with DMA). When the hardware is finished transferring each *blksize* sized block, the function *intr* should be called with the argument *intrarg* (typically from the audio hardware interrupt service routine). Once started the transfer may be stopped using *halt_output*.

Return 0 on success, otherwise an error code.

`int trigger_input(void *hdl, void *start, void *end,`
`int blksize, void (*intr)(void*), void *intrarg,`

`const audio_params_t *param)`

optional, is called to start the transfer of data from the audio hardware, parameterized as in *param*, to the circular buffer delimited by *start* and *end*. The call should return when the data transfer has been initiated (normally with DMA). When the hardware is finished transferring each *blksize* sized block, the function *intr* should be called with the argument *intrarg* (typically from the audio hardware interrupt service routine). Once started the transfer may be stopped using *halt_input*. Return 0 on success, otherwise an error code.

`int dev_ioctl(void *hdl, u_long cmd, void *addr,`

`int flag, struct lwp *l)`

optional, is called when an `ioctl`(2) is not recognized by the generic audio driver. Return 0 on success, otherwise an error code.

`int powerstate(void *hdl, int state)`

optional, is called on the first open and last close of the audio device. *state* may be one of `AUDIOPOWER_ON` or `AUDIOPOWER_OFF`. Returns 0 on success, otherwise an error code.

The *query_devinfo* method should define certain mixer controls for `AUDIO_SETINFO` to be able to change the port and gain, and `AUDIO_GETINFO` to read them, as follows.

If the record mixer is capable of input from more than one source, it should define `AudioNsource` in class `AudioCrecord`. This mixer control should be of type `AUDIO_MIXER_ENUM` or `AUDIO_MIXER_SET` and enumerate the possible input sources. Each of the named sources for which the recording level can be set should have a control in the `AudioCrecord` class of type `AUDIO_MIXER_VALUE`, except the "mixerout" source is special, and will never have its own control. Its selection signifies, rather, that various sources in class `AudioCrecord` will be combined and presented to the single recording output in the same fashion that the sources of class `AudioCinputs` are combined and presented to the playback output(s). If the overall recording level can be changed, regardless of the input source, then this control should be named `AudioNmaster` and be of class `AudioCrecord`.

Controls for various sources that affect only the playback output, as opposed to recording, should be in the `AudioCinputs` class, as of course should any controls that affect both playback and recording.

If the play mixer is capable of output to more than one destination, it should define `AudioNselect` in class `AudioCoutputs`. This mixer control should be of type `AUDIO_MIXER_ENUM` or `AUDIO_MIXER_SET` and enumerate the possible destinations. For each of the named destinations for which the output level can be set, there should be a control in the `AudioCoutputs` class of type `AUDIO_MIXER_VALUE`. If the overall output level can be changed, which is invariably the case, then this control should be named `AudioNmaster` and be of class `AudioCoutputs`.

There's one additional source recognized specially by `AUDIO_SETINFO` and `AUDIO_GETINFO`, to be presented as monitor_gain, and that is a control named `AudioNmonitor`, of class `AudioCmonitor`.

SEE ALSO

`audio`(4)

HISTORY

This **`audio`** interface first appeared in NetBSD 1.3.

NAME

autoconf, **config_search_loc**, **config_search_ia**, **config_found_sm_loc**, **config_found_ia**, **config_found**, **config_match**, **config_attach_loc**, **config_attach**, **config_attach_pseudo**, **config_detach**, **config_activate**, **config_deactivate**, **config_defer**, **config_interrupts**, **config_pending_incr**, **config_pending_decr**, **config_finalize_register** — autoconfiguration framework

SYNOPSIS

```
#include <sys/param.h>
#include <sys/device.h>
#include <sys/errno.h>

cfdata_t
config_search_loc(cfsubmatch_t func, device_t parent, const char *ia,
    const int *locs, void *aux);

cfdata_t
config_search_ia(cfsubmatch_t func, device_t parent, const char *ia,
    void *aux);

device_t
config_found_sm_loc(device_t parent, const char *ia, const int *locs,
    void *aux, cfprint_t print, cfsubmatch_t submatch);

device_t
config_found_ia(device_t parent, const char *ia, void *aux,
    cfprint_t print);

device_t
config_found(device_t parent, void *aux, cfprint_t print);

int
config_match(device_t parent, cfdata_t cf, void *aux);

device_t
config_attach_loc(device_t parent, cfdata_t cf, const int *locs, void *aux,
    cfprint_t print);

device_t
config_attach(device_t parent, cfdata_t cf, void *aux, cfprint_t print);

device_t
config_attach_pseudo(cfdata_t cf);

int
config_detach(device_t dev, int flags);

int
config_activate(device_t dev);

int
config_deactivate(device_t dev);

int
config_defer(device_t dev, void (*func)(device_t));

void
config_interrupts(device_t dev, void (*func)(device_t));
```

void
config_pending_incr();

void
config_pending_decr();

int
config_finalize_register(*device_t dev*, *int (*func)(device_t)*);

DESCRIPTION

Autoconfiguration is the process of matching hardware devices with an appropriate device driver. In its most basic form, autoconfiguration consists of the recursive process of finding and attaching all devices on a bus, including other busses.

The autoconfiguration framework supports *direct configuration* where the bus driver can determine the devices present. The autoconfiguration framework also supports *indirect configuration* where the drivers must probe the bus looking for the presence of a device. Direct configuration is preferred since it can find hardware regardless of the presence of proper drivers.

The autoconfiguration process occurs at system bootstrap and is driven by a table generated from a "machine description" file by `config`(1). For a description of the `config`(1) "device definition" language, see `config`(9).

Each device must have a name consisting of an alphanumeric string that ends with a unit number. The unit number identifies an instance of the driver. Device data structures are allocated dynamically during autoconfiguration, giving a unique address for each instance.

FUNCTIONS

config_search_loc(*func*, *parent*, *ia*, *locs*, *aux*)

Performs indirect configuration of physical devices. **config_search_loc**() iterates over all potential children, calling the given function *func* for each one. If *func* is `NULL`, **config_search_loc**() applies each child's match function instead. The argument *parent* is the pointer to the parent's device structure. The argument *ia* is the interface attribute on which the potential children should attach. It can be `NULL`, in which case all children attaching to any attribute are considered. The *locs* argument lists the locator values for the device and are passed to function *func*. The given *aux* argument describes the device that has been found and is simply passed on through *func* to the child. **config_search_loc**() returns a pointer to the best-matched child or `NULL` otherwise.

The role of *func* is to call the match function for each device and call **config_attach_loc**() for any positive matches. If *func* is `NULL`, then the parent should record the return value from **config_search_loc**() and call **config_attach_loc**() itself.

Note that this function is designed so that it can be used to apply an arbitrary function to all potential children. In this case callers may choose to ignore the return value.

config_search_ia(*func*, *parent*, *ia*, *aux*)

This function is equivalent to calling **config_search_loc**(*func*, *parent*, *ia*, *locs*, *aux*) with *locs* set to `NULL`.

config_found_sm_loc(*parent*, *ia*, *locs*, *aux*, *print*, *submatch*)

Performs direct configuration on a physical device. **config_found_sm_loc**() is called by the parent and in turn calls the *submatch* function to call the match function as determined by the configuration table. If *submatch* is `NULL`, the driver match functions are called directly. The argument *parent* is the pointer to the parent's device structure. The argument *ia* is the

name of the interface attribute on which the child will attach, per config(5) syntax. The argument *locs* lists the locator values for the device. The given *aux* argument describes the device that has been found. **config_found_sm_loc**() internally uses **config_search_loc**(), passing on *submatch*, *ia*, *locs* and *aux*. The *softc* structure for the matched device will be allocated, and the appropriate driver attach function will be called. If the device is matched, the system prints the name of the child and parent devices, and then calls the *print* function to produce additional information if desired. If no driver takes a match, the same *print* function is called to complain. The print function is called with the *aux* argument and, if the matches failed, the full name (including unit number) of the parent device, otherwise NULL. The *print* function must return an integer value.

Two special strings, "not configured" and "unsupported" will be appended automatically to non-driver reports if the return value is UNCONF or UNSUPP respectively; otherwise the function should return the value QUIET.

config_found_sm_loc() returns a pointer to the attached device's *softc* structure if the device is attached, NULL otherwise. Most callers can ignore this value, since the system will already have printed a diagnostic.

config_found_ia(*parent*, *ia*, *aux*, *print*)
This function is equivalent to calling **config_found_sm_loc**(*parent*, *ia*, *locs*, *aux*, *print*, *submatch*) with *locs* and *submatch* set to NULL. It is provided for better source code readability with locator-less device buses.

config_found(*parent*, *aux*, *print*)
This function is equivalent to calling **config_found_sm_loc**(*parent*, *ia*, *locs*, *aux*, *print*, *submatch*) with *ia*, *locs* and *submatch* set to NULL and is provided for compatibility with older drivers. New code should either make the interface attribute explicit or prefer an indirect method based on **config_search_loc**().

config_match(*parent*, *cf*, *aux*)
Match a device. Invokes the drivers match function according to the configuration table. The **config_match**() function returns a nonzero integer indicating the confidence of supporting this device and a value of 0 if the driver doesn't support the device.

config_attach_loc(*parent*, *locs*, *cf*, *aux*, *print*)
Attach a found device. Allocates the memory for the *softc* structure and calls the drivers attach function according to the configuration table. If successful, **config_attach_loc**() returns the *softc*. If unsuccessful, it returns NULL.

config_attach(*parent*, *cf*, *aux*, *print*)
This function is equivalent to calling **config_attach_loc**(*parent*, *cf*, *locs*, *aux*, *print*) with *locs* set to NULL.

config_attach_pseudo(*cf*)
Create an instance of a pseudo-device driver. config(5) syntax allows the creation of pseudo-devices from which regular *device_t* instances can be created. Such objects are similar to the devices that attach at the root of the device tree.

The caller is expected to allocate and fill the *cfdata_t* object and pass it to **config_attach_pseudo**(). The content of that object is similar to what is returned by **config_search_loc**() for regular devices.

config_detach(*dev*, *flags*)
Called by the parent to detach the child device. The second argument *flags* contains detachment flags. Valid values are DETACH_FORCE (force detachment (e.g., because of hardware removal)) and DETACH_QUIET (do not print a notice). **config_detach**() returns zero if

successful and an error code otherwise. **config_detach**() is always called from a thread context, allowing condition variables to be used while the device detaches itself.

config_activate(*dev*)
Called by the parent to activate the child device *dev*. It is called to activate resources and initialise other kernel subsystems (such as the network subsystem). **config_activate**() is called from interrupt context after the device has been attached.

config_deactivate(*dev*)
Called by the parent to deactivate the child device *dev*. **config_deactivate**() is called from interrupt context to immediately relinquish resources and notify dependent kernel subsystems that the device is about to be detached. At some later point **config_detach**() will be called to finalise the removal of the device.

config_defer(*dev*, *func*)
Called by the child to defer the remainder of its configuration until all its parent's devices have been attached. At this point, the function *func* is called with the argument *dev*.

config_interrupts(*dev*, *func*)
Called by the child to defer the remainder of its configuration until interrupts are enabled. At this point, the function *func* is called with the argument *dev*.

config_pending_incr()
Increment the *config_pending* semaphore. It is used to account for deferred configurations before mounting the root file system.

config_pending_decr()
Decrement the *config_pending* semaphore. It is used to account for deferred configurations before mounting the root file system.

config_finalize_register(*dev*, *func*)
Register a function to be called after all real devices have been found.

Registered functions are all executed until all of them return 0. The callbacks should return 0 to indicate they do not require to be called another time, but they should be aware that they still might be in case one of them returns 1.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the autoconfiguration framework can be found. All pathnames are relative to `/usr/src`.

The autoconfiguration framework itself is implemented within the file `sys/kern/subr_autoconf.c`. Data structures and function prototypes for the framework are located in `sys/sys/device.h`.

SEE ALSO

`config`(1), `config`(5), `condvar`(9), `config`(9), `driver`(9)

HISTORY

Autoconfiguration first appeared in 4.1BSD. The autoconfiguration framework was completely revised in 4.4BSD. The detach and activate/deactivate interfaces appeared in NetBSD 1.5.

NAME

bcdtobin, **bintobcd** — convert a single byte between (unsigned) packed bcd and binary

SYNOPSIS

```
#include <sys/systm.h>

unsigned int
bcdtobin(unsigned int bcd);

unsigned int
bintobcd(unsigned int bin);
```

DESCRIPTION

The **bcdtobin**() and **bintobcd**() functions convert a single byte between (unsigned) packed bcd and binary encodings.

RETURN VALUES

The **bcdtobin**() function returns the binary value of its BCD-encoded argument, *bcd*. The **bintobcd**() function returns the BCD encoding of its binary argument, *bin*.

NAME

bcmp — compare byte string

SYNOPSIS

```
#include <sys/systm.h>

int
bcmp(const void *b1, const void *b2, size_t len);
```

DESCRIPTION

The bcmp() interface is obsolete. Do not add new code using it. It will soon be purged. Use memcmp(9) instead. (The bcmp() function is now a macro for memcmp(9).)

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(9)

NAME

bcopy — copy byte string

SYNOPSIS

```
#include <sys/systm.h>

void
bcopy(const void *src, void *dst, size_t len);
```

DESCRIPTION

The bcopy() interface is obsolete. Do not add new code using it. It will soon be purged. Use memcpy(9) instead. (The bcopy() function is now a macro for memcpy(9).)

The **bcopy**() function copies *len* bytes from string *src* to string *dst*.

Unlike bcopy(3) the two strings must not overlap! In the traditional BSD kernel, overlapping copies were handled by the now-purged **ovbcopy**() function. If you need to copy overlapping data, see memmove(9).

If *len* is zero, no bytes are copied.

SEE ALSO

bcopy(3), memcpy(9), memmove(9)

NAME

bios32_service — locate BIOS32 service

SYNOPSIS

```
#include <i386/bios32.h>

int
bios32_service(uint32_t service, bios32_entry_t e, bios32_entry_info_t ei);
```

DESCRIPTION

The **bios32_service**() function calls the BIOS32 to locate the specified BIOS32 service *service* and fills in the entry point information *e* and *ei*.

SEE ALSO

bioscall(9)

NAME

bioscall — call system BIOS function from real mode

SYNOPSIS

```
#include <i386/bioscall.h>

void
bioscall(int function, struct bioscallregs *regs);
```

DESCRIPTION

The **bioscall** function switches the processor into real mode, calls the BIOS interrupt numbered *function*, and returns to protected mode.

This function is intended to be called during the initial system bootstrap when necessary to probe devices or pseudo-devices.

The register values specified by **regs* (with one exception) are installed before the BIOS interrupt is called. The processor flags are handled specially. Only the following flags are passed to the BIOS from the registers in *regs* (the remainder come from the processor's flags register at the time of the call): *PSL_C*, *PSL_PF*, *PSL_AF*, *PSL_Z*, *PSL_N*, *PSL_D*, *PSL_V*.

The *bioscallregs* structure is defined to contain structures for each register, to allow access to 32-, 16- or 8-bit wide sections of the registers. Definitions are provided which simplify access to the union members.

RETURN VALUES

bioscall fills in **regs* with the processor registers as returned from the BIOS call.

EXAMPLES

The Advanced Power Management driver calls **bioscall** by setting up a register structure with the APM installation check and device types in registers *ax* and *bx*, then calls the BIOS to fetch the details for calling the APM support through a protected-mode interface. The BIOS returns these details in the registers:

```
#include <i386/bioscall.h>
#include <i386/apmvar.h>
struct bioscallregs regs;

regs.AX = APM_BIOS_FN(APM_INSTALLATION_CHECK);
regs.BX = APM_DEV_APM_BIOS;
regs.CX = regs.DX = 0;
regs.ESI = regs.EDI = regs.EFLAGS = 0;
bioscall(APM_SYSTEM_BIOS, &regs);
```

CODE REFERENCES

sys/arch/i386/i386/bioscall.s, sys/arch/i386/bioscall/biostramp.S

REFERENCES

apm(4)

HISTORY

bioscall first appeared in NetBSD 1.3.

BUGS

Not all BIOS functions are safe to call through the trampoline, as they may depend on system state which has been disturbed or used for other purposes once the NetBSD kernel is running.

NAME

bitmask_snprintf — bitmask output conversion

SYNOPSIS

```
#include <sys/systm.h>

char *
bitmask_snprintf(u_quad_t val, const char *fmt, char *buf, size_t buflen);
```

DESCRIPTION

The **bitmask_snprintf**() 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. It follows two possible syntaxes, referred to as "old" and "new". The main advantage of the "new" formatting is that it is capable of handling multi-bit fields.

The first character of *fmt* may be `\177`, indicating that the remainder of the format string follows the "new" syntax. The second character (the first for the old format) 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. From here the syntaxes diverge.

The "old" format syntax is series of bit-position–description pairs. Each 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.

For the "new" format syntax, a bit-position–description begins with a field type followed by a binary bit-position and possibly a field length. The least significant bit is bit-position zero, unlike the "old" syntax where it is one.

b\B Describes a bit position. The bit-position *B* indicates the corresponding bit, as in the "old" format.

f\B\L Describes a multi-bit field beginning at bit-position *B* and having a bit-length of *L*. The remaining characters are printed as a description of the field followed by '=' and the value of the field. The value of the field is printed in the base specified as the second character of the decoding directive string *fmt*.

F\B\L Describes a multi-bit field like 'f', but just extracts the value for use with the '=' and ':' formatting directives described below.

=\V The field previously extracted by the last 'f' or 'F' operator is compared to the byte '**V**' (for values 0 through 255). If they are equal, '=' followed by the string following '**V**' is printed. This and the ':' operator may be repeated to annotate multiple possible values.

:\V Operates like the '=' operator, but omits the leading '='.

Finally, each field is delimited by a NUL ('\0') character. By convention, the format string has an additional NUL character at the end, following that delimiting the last bit-position–description pair.

The buffer *buf* passed to **bitmask_snprintf** must be at least KPRINTF_BUFSIZE bytes in length. See the source code for the definition of this macro.

RETURN VALUES

The **bitmask_snprintf**() function returns the buffer *buf*. The returned string is always NULL-terminated.

EXAMPLES

Two examples of the old formatting style:

```
bitmask_snprintf(3, "\10\2BITTWO\1BITONE", buf, buflen)
⇒ "3<BITTWO,BITONE>"

bitmask_snprintf(0xe860,
       "\20\x10NOTBOOT\x0fFPP\x0eSDVMA\x0cVIDEO"
       "\x0bLORES\x0aFPA\x09DIAG\x07CACHE"
       "\x06IOCACHE\x05LOOPBACK\x04DBGCACHE",
        buf, buflen)
⇒ "e860<NOTBOOT,FPP,SDVMA,VIDEO,CACHE,IOCACHE>"
```

An example of the new formatting style:

```
bitmask_snprintf(0x800f0701,
       "\177\020b\0LSB\0b\1_BITONE\0f\4\4NIBBLE2\0"
       "f\x10\4BURST\0=\4FOUR\0=\xfSIXTEEN\0"
       "b\x1fMSB\0\0",
        buf, buflen)
⇒ "800f0701<LSB,NIBBLE2=0,BURST=f=SIXTEEN,MSB>"
```

ERRORS

If the buffer *buf* is too small to hold the formatted output, **bitmask_snprintf**() will still return the buffer, containing a truncated string.

SEE ALSO

printf(9)

CODE REFERENCES

sys/kern/subr_prf_bitmask.c

HISTORY

The **bitmask_snprintf**() function was originally implemented as a non-standard %b format string for the kernel **printf**() function in NetBSD 1.5 and earlier releases.

AUTHORS

The "new" format was the invention of Chris Torek.

NAME

BLUETOOTH — Bluetooth Device/Protocol API

SYNOPSIS

```
#include <netbt/bluetooth.h>
#include <netbt/hci.h>
#include <netbt/l2cap.h>
#include <netbt/rfcomm.h>

struct hci_unit *
hci_attach(const struct hci_if *hci_if, device_t dev, uint16_t flags);

void
hci_detach(struct hci_unit *unit);

void
hci_input_event(struct hci_unit *unit, struct mbuf *m);

void
hci_input_acl(struct hci_unit *unit, struct mbuf *m);

void
hci_input_sco(struct hci_unit *unit, struct mbuf *m);

int
btproto_attach(btproto_handle *, const struct btproto *proto, void *ref);

int
btproto_bind(btproto_handle, struct sockaddr_bt *addr);

int
btproto_sockaddr(btproto_handle, struct sockaddr_bt *addr);

int
btproto_connect(btproto_handle, struct sockaddr_bt *addr);

int
btproto_peeraddr(btproto_handle, struct sockaddr_bt *addr);

int
btproto_disconnect(btproto_handle, int linger);

int
btproto_detach(btproto_handle *);

int
btproto_listen(btproto_handle);

int
btproto_send(btproto_handle, struct mbuf *mbuf);

int
btproto_rcvd(btproto_handle, size_t space);

int
btproto_setopt(btproto_handle, int optarg, void *arg);

int
btproto_getopt(btproto_handle, int optarg, void *arg);
```

DESCRIPTION

The Bluetooth Protocol Stack provides socket based access to Bluetooth Devices. This document describes device driver access to the stack from below, and also the general Bluetooth Protocol/Service API for layering above existing Bluetooth Protocols.

DATA TYPES

Device drivers attaching to the Bluetooth Protocol Stack should pass a pointer to a *struct hci_if* defined in <netbt/hci.h> containing the driver information as follows:

```
struct hci_if {
        int     (*enable)(device_t);
        void    (*disable)(device_t);
        void    (*output_cmd)(device_t, struct mbuf *);
        void    (*output_acl)(device_t, struct mbuf *);
        void    (*output_sco)(device_t, struct mbuf *);
        void    (*get_stats)(device_t, struct bt_stats *, int);
        int     ipl;
};
```

Statistics counters should be updated by the device after packets have been transmitted or received, or when errors occur.

```
struct bt_stats {
        uint32_t        err_tx;
        uint32_t        err_rx;
        uint32_t        cmd_tx;
        uint32_t        evt_rx;
        uint32_t        acl_tx;
        uint32_t        acl_rx;
        uint32_t        sco_tx;
        uint32_t        sco_rx;
        uint32_t        byte_tx;
        uint32_t        byte_rx;
};
```

Bluetooth Protocol layers attaching above the Bluetooth Protocol Stack will make use of the *struct btproto* data type, which is defined in <netbt/bluetooth.h> and contains the following function callbacks which should be initialized by the protocol layer before attaching to the protocol which it uses:

```
struct btproto {
        void (*connecting)(void *);
        void (*connected)(void *);
        void (*disconnected)(void *, int);
        void *(*newconn)(void *, struct sockaddr_bt *, struct sockaddr_bt *);
        void (*complete)(void *, int);
        void (*linkmode)(void *, int);
        void (*input)(void *, struct mbuf *);
};
```

FUNCTIONS

The following functions are related to the Bluetooth Device API.

hci_attach(*hci_if*, *dev*)

Attach Bluetooth HCI device *dev* to the protocol stack in the manner described by *hci_if*. Driver quirks may be registered by passing the corresponding BTF_xxxx flag in the *flags* argument.

hci_attach() will return a *struct hci_unit* handle to be passed to the protocol stack in other calls.

hci_detach(*unit*)
Detach Bluetooth HCI *unit* from the device.

hci_input_event(*unit*, *mbuf*)
This function should be called by the device when it has an event packet to present to the protocol stack. It may be called from an interrupt routine at the *ipl* value given in the *hci_if* descriptor.

hci_input_acl(*unit*, *mbuf*)
This function should be called by the device when it has an ACL data packet to present to the protocol stack. It may be called from an interrupt routine at the *ipl* value given in the *hci_if* descriptor.

hci_input_sco(*unit*, *mbuf*)
This function should be called by the device when it has an SCO data packet to present to the protocol stack. It may be called from an interrupt routine at the *ipl* value given in the *hci_if* descriptor.

(*enable)(*dev*)
This will be called when the protocol stack wishes to enable the device.

(*disable)(*dev*)
This will be called when the protocol stack wishes to disable the device.

(*output_cmd)(*dev*, *mbuf*)
Will be called to output command packets on the device. The device is responsible for arbitrating access to the output queue, and output commands should be sent asynchronously. The device owns the *mbuf* and should release it when sent.

(*output_acl)(*dev*, *mbuf*)
Will be called to output ACL data packets on the device. The device is responsible for arbitrating access to the output queue, and ACL data packets should be sent asynchronously. The device owns the *mbuf* and should release it when sent.

(*output_sco)(*dev*, *mbuf*)
Will be called to output SCO data packets on the device. The device is responsible for arbitrating access to the output queue, and SCO data packets should be sent asynchronously. When the SCO data packet has been placed on the device and the *mbuf* is no longer required, it should be returned to the Bluetooth protocol stack via the **hci_complete_sco**() call.

(*get_stats)(*dev*, *dest*, *flush*)
Will be called when IO statistics are requested. The *bt_stats* structure *dest* should be filled in, and if the *flush* argument is true, statistics should be reset.

The following function definitions are related to the Bluetooth Protocol API. Note that the "btproto" prefix is representative only, the protocol being used will have a more specific prefix with prototypes being declared in the appropriate <netbt/btproto.h> file.

btproto_attach(*handle_ptr*, *proto*, *ref*)
Allocate and initialize a new protocol object at the *handle_ptr* address that should subsequently be passed into the other functions. *proto* is a pointer to the *btproto* structure as described above containing relevant callbacks, and *ref* is the argument that will be supplied to those calls.

btproto_bind(*handle*, *addr*)
Set the local address of the protocol object described by *handle* to *addr*.

btproto_sockaddr(*handle*, *addr*)
Copy the local address of the protocol object described by *handle* into *addr*

btproto_connect(*handle*, *addr*)
Initiate a connection by the protocol object described by *handle* to the remote device described by *addr*. This will result in a call to either **proto->connected**() or **proto->disconnected**(), and optionally **proto->connecting**() with the appropriate reference as given to **btproto_attach**().

btproto_peeraddr(*handle*, *addr*)
Copy the remote address of the protocol object described by *handle* into *addr*.

btproto_disconnect(*handle*, *linger*)
Schedule a disconnection by the protocol object described by *handle*. This will result in a call to **proto->disconnected**() with the appropriate reference when the connection is torn down. If linger is zero, the disconnection will be initiated immediately and any outstanding data may be lost.

btproto_detach(*handle_ptr*)
Detach the protocol object described by the value in the location of *handle_ptr*, and free any related memory. The pointer in the location is cleared.

btproto_listen(*handle*)
Use the protocol object described by *handle* as a listening post. This will result in calls to the **proto->newconn**() function when incoming connections are detected.

btproto_send(*handle*, *mbuf*)
Send data on the connection described by the protocol object.

btproto_rcvd(*handle*, *space*)
Indicate to the protocol that *space* is now available in the input buffers so that flow control may be deasserted. This should also be called to indicate initial buffer space. Note that *space* is an absolute value.

btproto_setopt(*handle*, *optarg*, *arg*)
Set options on the protocol object described by *handle*.

btproto_getopt(*handle*, *optarg*, *arg*)
Get options for the protocol object described by *handle*.

(*connecting)(*ref*)
This function will be called when the protocol receives information that the connection described by *ref* is pending.

(*connected)(*ref*)
This function will be called when the connection described by *ref* is successful and indicates that data may now be sent.

(*disconnected)(*ref*, *error*)
This function will be called when the connection described by *ref* is disconnected.

***(*newconn)**(*ref*, *laddr*, *raddr*)
This function will be called when the protocol receives a new incoming connection on the local device described by *laddr* from the remote device described by *raddr*. The protocol should decide if it wishes to accept the connection and should attach and return a new instance of the relevant protocol handle or NULL.

(*complete)(*ref*, *count*)
This function will be called when the protocol has completed sending data. Complete will usually mean that the data has successfully left the device though for guaranteed protocols it can mean that the data has arrived at the other end and been acknowledged, and that *count* amount of data can be removed from the socket buffer. The units of the *count* value will be dependent on the protocol

being used (e.g. RFCOMM is bytes, but L2CAP is packets)

(*linkmode)(*ref*, *mode*)
This function will be called for established connections, when the link mode of the baseband link has changed. *mode* is the new mode.

(*input)(*ref*, *mbuf*)
This function is called to supply new data on the connection described by *ref*.

CODE REFERENCES

This section describes places in the NetBSD source tree where actual code implementing or using the Bluetooth Protocol Stack can be found. All pathnames are relative to `/usr/src`.

The Bluetooth Protocol Stack is contained in the `sys/netbt` directory.

The Bluetooth Device API as described above is contained in the `sys/netbt/hci_unit.c` file.

For examples of the Bluetooth Protocol API see the interaction between the L2CAP upper layer in `sys/netbt/l2cap_upper.c` and either the L2CAP socket layer in `sys/netbt/l2cap_socket.c` or the `bthidev`(4) pseudo-device in `sys/dev/bluetooth/bthidev.c`.

Also, the RFCOMM upper layer in `sys/netbt/rfcomm_upper.c` and the RFCOMM socket layer in `sys/netbt/rfcomm_socket.c`.

SEE ALSO

`bluetooth`(4), `bt3c`(4), `bthidev`(4), `ubt`(4)

HISTORY

This Bluetooth Protocol Stack was written for NetBSD 4.0 by Iain Hibbert, under the sponsorship of Itronix, Inc.

NAME

buffercache, **bread**, **breada**, **breadn**, **bwrite**, **bawrite**, **bdwrite**, **getblk**, **geteblk**, **incore**, **allocbuf**, **brelse**, **biodone**, **biowait** — buffer cache interfaces

SYNOPSIS

```
#include <sys/buf.h>

int
bread(struct vnode *vp, daddr_t blkno, int size, struct kauth_cred *cred,
    int flags, struct buf **bpp);

int
breadn(struct vnode *vp, daddr_t blkno, int size, daddr_t rablks[],
    int rasizes[], int nrablks, struct kauth_cred *cred, int flags,
    struct buf **bpp);

int
breada(struct vnode *vp, daddr_t blkno, int size, daddr_t rablkno,
    int rabsize, struct kauth_cred *cred, int flags, struct buf **bpp);

int
bwrite(struct buf *bp);

void
bawrite(struct buf *bp);

void
bdwrite(struct buf *bp);

struct buf *
getblk(struct vnode *vp, daddr_t blkno, int size, int slpflag, int slptimeo);

struct buf *
geteblk(int size);

struct buf *
incore(struct vnode *vp, daddr_t blkno);

void
allocbuf(struct buf *bp, int size, int preserve);

void
brelse(struct buf *bp);

void
biodone(struct buf *bp);

int
biowait(struct buf *bp);
```

DESCRIPTION

The **buffercache** interface is used by each filesystems to improve I/O performance using in-core caches of filesystem blocks.

The kernel memory used to cache a block is called a buffer and described by a *buf* structure. In addition to describing a cached block, a *buf* structure is also used to describe an I/O request as a part of the disk driver interface.

FUNCTIONS

bread(*vp*, *blkno*, *size*, *cred*, *flags*, *bpp*)

Read a block corresponding to *vp* and *blkno*. The buffer is returned via *bpp*. The units of *blkno* are specifically the units used by the **VOP_STRATEGY**() routine for the *vp* vnode. For device special files, *blkno* is in units of DEV_BSIZE and both *blkno* and *size* must be multiples of the underlying device's block size. For other files, *blkno* is in units chosen by the file system containing *vp*.

If the buffer is not found (i.e. the block is not cached in memory), **bread**() allocates a buffer with enough pages for *size* and reads the specified disk block into it using credential *cred*.

The buffer returned by **bread**() is marked as busy. (The B_BUSY flag is set.) After manipulation of the buffer returned from **bread**(), the caller should unbusy it so that another thread can get it. If the buffer contents are modified and should be written back to disk, it should be unbusied using one of variants of **bwrite**(). Otherwise, it should be unbusied using **brelse**().

breadn(*vp*, *blkno*, *size*, *rablks*, *rasizes*, *nrablks*, *cred*, *flags*, *bpp*)

Get a buffer as **bread**(). In addition, **breadn**() will start read-ahead of blocks specified by *rablks*, *rasizes*, *nrablks*.

breada(*vp*, *blkno*, *size*, *rablkno*, *rabsize*, *cred*, *flags*, *bpp*)

Same as **breadn**() with single block read-ahead. This function is for compatibility with old filesystem code and shouldn't be used by new ones.

bwrite(*bp*)

Write a block. Start I/O for write using **VOP_STRATEGY**(). Then, unless the B_ASYNC flag is set in *bp*, **bwrite**() waits for the I/O to complete.

bawrite(*bp*)

Write a block asynchronously. Set the B_ASYNC flag in *bp* and simply call **VOP_BWRITE**(), which results in **bwrite**() for most filesystems.

bdwrite(*bp*)

Delayed write. Unlike **bawrite**(), **bdwrite**() won't start any I/O. It only marks the buffer as dirty (B_DELWRI) and unbusy it.

getblk(*vp*, *blkno*, *size*, *slpflag*, *slptimeo*)

Get a block of requested size *size* that is associated with a given vnode and block offset, specified by *vp* and *blkno*. If it is found in the block cache, make it busy and return it. Otherwise, return an empty block of the correct size. It is up to the caller to ensure that the cached blocks are of the correct size.

If **getblk**() needs to sleep, *slpflag* and *slptimeo* are used as arguments for **cv_timedwait**().

geteblk(*size*)

Allocate an empty, disassociated block of a given size *size*.

incore(*vp*, *blkno*)

Determine if a block associated to a given vnode and block offset is in the cache. If it is there, return a pointer to it. Note that **incore**() doesn't busy the buffer unlike **getblk**().

allocbuf(*bp*, *size*, *preserve*)

Expand or contract the actual memory allocated to a buffer. If *preserve* is zero, the entire data in the buffer will be lost. Otherwise, if the buffer shrinks, the truncated part of the data is lost, so it is up to the caller to have written it out *first* if needed; this routine will not start a write. If the buffer grows, it is the callers responsibility to fill out the buffer's additional contents.

brelse(*bp*)
Unbusy a buffer and release it to the free lists.

biodone(*bp*)
Mark I/O complete on a buffer. If a callback has been requested by B_CALL, do so. Otherwise, wakeup waiters.

biowait(*bp*)
Wait for operations on the buffer to complete. When they do, extract and return the I/O's error value.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing the buffer cache subsystem can be found. All pathnames are relative to /usr/src.

The buffer cache subsystem is implemented within the file sys/kern/vfs_bio.c.

SEE ALSO

intro(9), vnode(9)

Maurice J. Bach, *The Design of the UNIX Operating System*, *Prentice Hall*, 1986.

Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, *Addison Wesley*, 1996.

BUGS

In the current implementation, **bread**() and its variants don't use a specified credential.

Because **biodone**() and **biowait**() do not really belong to **buffercache**, they shouldn't be documented here.

NAME

bufq, **bufq_state**, **bufq_alloc**, **bufq_drain**, **bufq_free**, **bufq_getstrategyname**, **bufq_move**, **BUFQ_PUT**, **BUFQ_GET**, **BUFQ_PEEK**, **BUFQ_CANCEL** — device buffer queues

SYNOPSIS

```
#include <sys/bufq.h>

int
bufq_alloc(struct bufq_state **bufq, const char *strategy, int flags);

void
bufq_drain(struct bufq_state *bufq);

void
bufq_free(struct bufq_state *bufq);

const char *
bufq_getstrategyname(struct bufq_state *bufq);

void
bufq_move(struct bufq_state *dst, struct bufq_state *src);

void
BUFQ_PUT(struct bufq_state *bufq, struct buf *bp);

struct buf *
BUFQ_GET(struct bufq_state *bufq);

struct buf *
BUFQ_PEEK(struct bufq_state *bufq);

struct buf *
BUFQ_CANCEL(struct bufq_state *bufq, struct buf *bp);
```

DESCRIPTION

The **bufq** subsystem is a set of operations for the management of device buffer queues.

The primary data type for using the operations is the *bufq_state* structure, which is opaque for users.

FUNCTIONS

bufq_alloc(*bufq*, *strategy*, *flags*)

Allocate and initialize a *bufq_state* descriptor.

The argument *strategy* specifies a buffer queue strategy to be used for this buffer queue. The following special values can be used:

BUFQ_STRAT_ANY	Let **bufq_alloc**() select a strategy.
BUFQ_DISK_DEFAULT_STRAT	Let **bufq_alloc**() select a strategy, assuming it will be used for a normal disk device.

Valid bits for the *flags* are:

BUFQ_SORT_RAWBLOCK	sort by *b_rawblkno*
BUFQ_SORT_CYLINDER	sort by *b_cylinder* and then by *b_rawblkno*
BUFQ_EXACT	Fail if a strategy specified by *strategy* is not available. In that case, *bufq_alloc* returns ENOENT. If this flag is not specified, **bufq_alloc**() will silently use one of available strategies.

bufq_drain(*bufq*)
Drain a *bufq_state* descriptor.

bufq_free(*bufq*)
Destroy a *bufq_state* descriptor.

bufq_getstrategyname(*bufq*)
Get a strategy identifier of a buffer queue, the string returned will be NUL-terminated and it always will be a valid strategy name.

bufq_move(*dst*, *src*)
Move all requests from the buffer queue *src* to the buffer queue *dst*.

BUFQ_PUT(*bufq*, *bp*)
Put the buf *bp* in the queue.

BUFQ_GET(*bufq*)
Get the next buf from the queue and remove it from the queue. Returns NULL if the queue is empty.

BUFQ_PEEK(*bufq*)
Get the next buf from the queue without removal. The next buf will remain the same until **BUFQ_GET**(), **BUFQ_PUT**(), or **bufq_drain**() is called. Returns NULL if the queue is empty.

BUFQ_CANCEL(*bufq*, *bp*)
Cancel the buf *bp* issued earlier on the queue. Returns NULL if the element can not be found on the queue or *bp* if it has been found and removed.

CODE REFERENCES

The actual code implementing the device buffer queues can be found in the file sys/kern/subr_bufq.c.

HISTORY

The **bufq** subsystem appeared in NetBSD 2.0.

AUTHORS

The **bufq** subsystem was written by Jürgen Hannken-Illjes ⟨hannken@NetBSD.org⟩.

NAME

bus_dma, **bus_dmamap_create**, **bus_dmamap_destroy**, **bus_dmamap_load**, **bus_dmamap_load_mbuf**, **bus_dmamap_load_uio**, **bus_dmamap_load_raw**, **bus_dmamap_unload**, **bus_dmamap_sync**, **bus_dmamem_alloc**, **bus_dmamem_free**, **bus_dmamem_map**, **bus_dmamem_unmap**, **bus_dmamem_mmap**, **bus_dmatag_subregion**, **bus_dmatag_destroy** — Bus and Machine Independent DMA Mapping Interface

SYNOPSIS

```
#include <machine/bus.h>

int
bus_dmamap_create(bus_dma_tag_t tag, bus_size_t size, int nsegments,
    bus_size_t maxsegsz, bus_size_t boundary, int flags,
    bus_dmamap_t *dmamp);

void
bus_dmamap_destroy(bus_dma_tag_t tag, bus_dmamap_t dmam);

int
bus_dmamap_load(bus_dma_tag_t tag, bus_dmamap_t dmam, void *buf,
    bus_size_t buflen, struct lwp *l, int flags);

int
bus_dmamap_load_mbuf(bus_dma_tag_t tag, bus_dmamap_t dmam,
    struct mbuf *chain, int flags);

int
bus_dmamap_load_uio(bus_dma_tag_t tag, bus_dmamap_t dmam, struct uio *uio,
    int flags);

int
bus_dmamap_load_raw(bus_dma_tag_t tag, bus_dmamap_t dmam,
    bus_dma_segment_t *segs, int nsegs, bus_size_t size, int flags);

void
bus_dmamap_unload(bus_dma_tag_t tag, bus_dmamap_t dmam);

void
bus_dmamap_sync(bus_dma_tag_t tag, bus_dmamap_t dmam, bus_addr_t offset,
    bus_size_t len, int ops);

int
bus_dmamem_alloc(bus_dma_tag_t tag, bus_size_t size, bus_size_t alignment,
    bus_size_t boundary, bus_dma_segment_t *segs, int nsegs, int *rsegs,
    int flags);

void
bus_dmamem_free(bus_dma_tag_t tag, bus_dma_segment_t *segs, int nsegs);

int
bus_dmamem_map(bus_dma_tag_t tag, bus_dma_segment_t *segs, int nsegs,
    size_t size, void **kvap, int flags);

void
bus_dmamem_unmap(bus_dma_tag_t tag, void *kva, size_t size);

paddr_t
bus_dmamem_mmap(bus_dma_tag_t tag, bus_dma_segment_t *segs, int nsegs,
```

off_t off, *int prot*, *int flags*);

int
bus_dmatag_subregion(*bus_dma_tag_t tag*, *bus_addr_t min_addr*, *bus_addr_t max_addr*, *bus_dma_tag_t *newtag*, *int flags*);

void
bus_dmatag_destroy(*bus_dma_tag_t tag*);

DESCRIPTION

Provide a bus- and machine-independent "DMA mapping interface."

NOTES

All data structures, function prototypes, and macros will be defined by the port-specific header ⟨machine/bus.h⟩. Note that this document assumes the existence of types already defined by the current "bus.h" interface.

Unless otherwise noted, all function calls in this interface may be defined as cpp(1) macros.

DATA TYPES

Individual implementations may name these structures whatever they wish, providing that the external representations are:

bus_dma_tag_t

A machine-dependent opaque type describing the implementation of DMA for a given bus.

bus_dma_segment_t

A structure with at least the following members:

```
bus_addr_t      ds_addr;
bus_size_t      ds_len;
```

The structure may have machine-dependent members and arbitrary layout. The values in *ds_addr* and *ds_len* are suitable for programming into DMA controller address and length registers.

bus_dmamap_t

A pointer to a structure with at least the following members:

```
bus_size_t      dm_maxsegsz;
bus_size_t      dm_mapsize;
int             dm_nsegs;
bus_dma_segment_t *dm_segs;
```

The structure may have machine-dependent members and arbitrary layout. The *dm_maxsegsz* member indicates the maximum number of bytes that may be transferred by any given DMA segment. The *dm_mapsize* member indicates the size of the mapping. A value of 0 indicates the mapping is invalid. The *dm_segs* member may be an array of segments or a pointer to an array of segments. The *dm_nsegs* member indicates the number of segments in *dm_segs*.

FUNCTIONS

bus_dmamap_create(*tag*, *size*, *nsegments*, *maxsegsz*, *boundary*, *flags*, *dmamp*)

Allocates a DMA handle and initializes it according to the parameters provided. Arguments are as follows:

tag This is the bus_dma_tag_t passed down from the parent driver via *<bus>_attach_args*.

size This is the maximum DMA transfer that can be mapped by the handle.

nsegments Number of segments the device can support in a single DMA transaction. This may be the number of scatter-gather descriptors supported by the device.

maxsegsz The maximum number of bytes that may be transferred by any given DMA segment and will be assigned to the *dm_maxsegsz* member.

boundary Some DMA controllers are not able to transfer data that crosses a particular boundary. This argument allows this boundary to be specified. The boundary lines begin at 0, and occur every *boundary* bytes. Mappings may begin on a boundary line but may not end on or cross a boundary line. If no boundary condition needs to be observed, a *boundary* argument of 0 should be used.

flags Flags are defined as follows:

- BUS_DMA_WAITOK It is safe to wait (sleep) for resources during this call.
- BUS_DMA_NOWAIT It is not safe to wait (sleep) for resources during this call.
- BUS_DMA_ALLOCNOW Perform any resource allocation this handle may need now. If this is not specified, the allocation may be deferred to **bus_dmamap_load**(). If this flag is specified, **bus_dmamap_load**() will not block on resource allocation.
- BUS_DMA_BUS[1-4] These flags are placeholders, and may be used by busses to provide bus-dependent functionality.

dmamp This is a pointer to a bus_dmamap_t. A DMA map will be allocated and pointed to by *dmamp* upon successful completion of this routine.

Behavior is not defined if invalid arguments are passed to **bus_dmamap_create**().

Returns 0 on success, or an error code to indicate mode of failure.

bus_dmamap_destroy(*tag*, *dmam*)

Frees all resources associated with a given DMA handle. Arguments are as follows:

tag This is the bus_dma_tag_t passed down from the parent driver via *<bus>_attach_args*.

dmam The DMA handle to destroy.

In the event that the DMA handle contains a valid mapping, the mapping will be unloaded via the same mechanism used by **bus_dmamap_unload**().

Behavior is not defined if invalid arguments are passed to **bus_dmamap_destroy**().

If given valid arguments, **bus_dmamap_destroy**() always succeeds.

bus_dmamap_load(*tag*, *dmam*, *buf*, *buflen*, *l*, *flags*)

Loads a DMA handle with mappings for a DMA transfer. It assumes that all pages involved in a DMA transfer are wired. Arguments are as follows:

tag This is the bus_dma_tag_t passed down from the parent driver via *<bus>_attach_args*.

dmam The DMA handle with which to map the transfer.

buf The buffer to be used for the DMA transfer.

buflen The size of the buffer.

l Used to indicate the address space in which the buffer is located. If NULL, the buffer is assumed to be in kernel space. Otherwise, the buffer is assumed to be in lwp *l*'s address space.

flags are defined as follows:

BUS_DMA_WAITOK	It is safe to wait (sleep) for resources during this call.
BUS_DMA_NOWAIT	It is not safe to wait (sleep) for resources during this call.
BUS_DMA_STREAMING	By default, the **bus_dma** API assumes that there is coherency between memory and the device performing the DMA transaction. Some platforms, however, have special hardware, such as an "I/O cache", which may improve performance of some types of DMA transactions, but which break the assumption that there is coherency between memory and the device performing the DMA transaction. This flag allows the use of this special hardware, provided that the device is doing sequential, unidirectional transfers which conform to certain alignment and size constraints defined by the platform. If the platform does not support the feature, or if the buffer being loaded into the DMA map does not conform to the constraints required for use of the feature, then this flag will be silently ignored. Also refer to the use of this flag with the **bus_dmamem_alloc**() function.
BUS_DMA_READ	This is a hint to the machine-dependent back-end that indicates the mapping will be used only for a *device -> memory* transaction. The back-end may perform optimizations based on this information.
BUS_DMA_WRITE	This is a hint to the machine-dependent back-end that indicates the mapping will be used only for a *memory -> device* transaction. The back-end may perform optimizations based on this information.
BUS_DMA_BUS[1-4]	These flags are placeholders, and may be used by busses to provide bus-dependent functionality.

As noted above, if a DMA handle is created with BUS_DMA_ALLOCNOW, **bus_dmamap_load**() will never block.

If a call to **bus_dmamap_load**() fails, the mapping in the DMA handle will be invalid. It is the responsibility of the caller to clean up any inconsistent device state resulting from incomplete iteration through the uio.

Behavior is not defined if invalid arguments are passed to **bus_dmamap_load**().

Returns 0 on success, or an error code to indicate mode of failure.

bus_dmamap_load_mbuf(*tag*, *dmam*, *chain*, *flags*)
This is a variation of **bus_dmamap_load**() which maps mbuf chains for DMA transfers. Mbuf chains are assumed to be in kernel virtual address space.

bus_dmamap_load_uio(*tag*, *dmam*, *uio*, *flags*)
This is a variation of **bus_dmamap_load**() which maps buffers pointed to by *uio* for DMA transfers. Determination if the buffers are in user or kernel virtual address space is done internally, according to *uio->uio_vmspace*. See uiomove(9) for details of the uio structure.

bus_dmamap_load_raw(*tag*, *dmam*, *segs*, *nsegs*, *size*, *flags*)
This is a variation of **bus_dmamap_load**() which maps buffers allocated by **bus_dmamem_alloc**() (see below). The *segs* argument is an array of bus_dma_segment_t's filled in by **bus_dmamem_alloc**(). The *nsegs* argument is the number of segments in the array. The *size* argument is the size of the DMA transfer.

bus_dmamap_unload(*tag*, *dmam*)

Deletes the mappings for a given DMA handle. Arguments are as follows:

tag This is the bus_dma_tag_t passed down from the parent driver via *<bus>_attach_args*.

dmam The DMA handle containing the mappings which are to be deleted.

If the DMA handle was created with BUS_DMA_ALLOCNOW, **bus_dmamap_unload**() will not free the corresponding resources which were allocated by **bus_dmamap_create**(). This is to ensure that **bus_dmamap_load**() will never block on resources if the handle was created with BUS_DMA_ALLOCNOW.

bus_dmamap_unload() will not perform any implicit synchronization of DMA buffers. This must be done explicitly by **bus_dmamap_sync**().

bus_dmamap_unload() will restore the *dm_maxsegsz* member to its initial value assigned by **bus_dmamap_create**().

Behavior is not defined if invalid arguments are passed to **bus_dmamap_unload**().

If given valid arguments, **bus_dmamap_unload**() always succeeds.

bus_dmamap_sync(*tag*, *dmam*, *offset*, *len*, *ops*)

Performs pre- and post-DMA operation cache and/or buffer synchronization. Arguments are as follows:

tag This is the bus_dma_tag_t passed down from the parent driver via *<bus>_attach_args*.

dmam The DMA mapping to be synchronized.

offset The offset into the DMA mapping to synchronize.

len The length of the mapping from *offset* to synchronize.

ops One or more synchronization operation to perform. The following DMA synchronization operations are defined:

- BUS_DMASYNC_PREREAD — Perform any pre-read DMA cache and/or bounce operations.
- BUS_DMASYNC_POSTREAD — Perform any post-read DMA cache and/or bounce operations.
- BUS_DMASYNC_PREWRITE — Perform any pre-write DMA cache and/or bounce operations.
- BUS_DMASYNC_POSTWRITE — Perform any post-write DMA cache and/or bounce operations.

More than one operation may performed in a given synchronization call. Mixing of *PRE* and *POST* operations is not allowed, and behavior is undefined if this is attempted.

Synchronization operations are expressed from the perspective of the host RAM, e.g., a *device -> memory* operation is a *READ* and a *memory -> device* operation is a *WRITE*.

bus_dmamap_sync() may consult state kept within the DMA map to determine if the memory is mapped in a DMA coherent fashion. If so, **bus_dmamap_sync**() may elect to skip certain expensive operations, such as flushing of the data cache. See **bus_dmamem_map**() for more information on this subject.

On platforms which implement a weak memory access ordering model, **bus_dmamap_sync**() will always cause the appropriate memory barriers to be issued.

This function exists to ensure that the host and the device have a consistent view of a range of DMA memory, before and after a DMA operation.

An example of using **bus_dmamap_sync**(), involving multiple read-write use of a single mapping might look like this:

```
bus_dmamap_load(...);

while (not done) {
        /* invalidate soon-to-be-stale cache blocks */
        bus_dmamap_sync(..., BUS_DMASYNC_PREREAD);

        [ do read DMA ]

        /* copy from bounce */
        bus_dmamap_sync(..., BUS_DMASYNC_POSTREAD);

        /* read data now in driver-provided buffer */

        [ computation ]

        /* data to be written now in driver-provided buffer */

        /* flush write buffers and writeback, copy to bounce */
        bus_dmamap_sync(..., BUS_DMASYNC_PREWRITE);

        [ do write DMA ]

        /* probably a no-op, but provided for consistency */
        bus_dmamap_sync(..., BUS_DMASYNC_POSTWRITE);
}

bus_dmamap_unload(...);
```

This function *must* be called to synchronize DMA buffers before and after a DMA operation. Other **bus_dma** functions can *not* be relied on to do this synchronization implicitly. If DMA read and write operations are not preceded and followed by the appropriate synchronization operations, behavior is undefined.

Behavior is not defined if invalid arguments are passed to **bus_dmamap_sync**().

If given valid arguments, **bus_dmamap_sync**() always succeeds.

bus_dmamem_alloc(*tag*, *size*, *alignment*, *boundary*, *segs*, *...*)
Allocates memory that is "DMA safe" for the bus corresponding to the given tag.

The mapping of this memory is machine-dependent (or "opaque"); machine-independent code is not to assume that the addresses returned are valid in kernel virtual address space, or that the addresses returned are system physical addresses. The address value returned as part of *segs* can thus not be used to program DMA controller address registers. Only the values in the *dm_segs* array of a successfully loaded DMA map (using **bus_dmamap_load**()) can be used for this purpose.

Allocations will always be rounded to the hardware page size. Callers may wish to take advantage of this, and cluster allocation of small data structures. Arguments are as follows:

tag This is the bus_dma_tag_t passed down from the parent driver via *<bus>_attach_args*.

size The amount of memory to allocate.

alignment Each segment in the allocated memory will be aligned to this value. If the alignment is less than a hardware page size, it will be rounded up to the hardware page size. This value must be a power of two.

boundary Each segment in the allocated memory must not cross this boundary (relative to zero). This value must be a power of two. A boundary value less than the size of the allocation is invalid.

segs An array of bus_dma_segment_t's, filled in as memory is allocated, representing the opaque addresses of the memory chunks.

nsegs Specifies the number of segments in *segs*, and this is the maximum number of segments that the allocated memory may contain.

rsegs Used to return the actual number of segments the memory contains.

flags Flags are defined as follows:

- `BUS_DMA_WAITOK` It is safe to wait (sleep) for resources during this call.
- `BUS_DMA_NOWAIT` It is not safe to wait (sleep) for resources during this call.
- `BUS_DMA_STREAMING` Adjusts, if necessary, the size, alignment, and boundary constrains to conform to the platform-dependent requirements for the use of the `BUS_DMA_STREAMING` flag with the **bus_dmamap_load**() function. If the platform does not support the `BUS_DMA_STREAMING` feature, or if the size, alignment, and boundary constraints would already satisfy the platform's requirements, this flag is silently ignored. The `BUS_DMA_STREAMING` flag will never relax the constraints specified in the call.
- `BUS_DMA_BUS[1-4]` These flags are placeholders, and may be used by busses to provide bus-dependent functionality.

All pages allocated by **bus_dmamem_alloc**() will be wired down until they are freed by **bus_dmamem_free**().

Behavior is undefined if invalid arguments are passed to **bus_dmamem_alloc**().

Returns 0 on success, or an error code indicating mode of failure.

bus_dmamem_free(*tag*, *segs*, *nsegs*)

Frees memory previously allocated by **bus_dmamem_alloc**(). Any mappings will be invalidated. Arguments are as follows:

tag This is the bus_dma_tag_t passed down from the parent driver via *<bus>_attach_args*.

segs The array of bus_dma_segment_t's filled in by **bus_dmamem_alloc**().

nsegs The number of segments in *segs*.

Behavior is undefined if invalid arguments are passed to **bus_dmamem_free**().

If given valid arguments, **bus_dmamem_free**() always succeeds.

bus_dmamem_map(*tag*, *segs*, *nsegs*, *size*, *kvap*, *flags*)

Maps memory allocated with **bus_dmamem_alloc**() into kernel virtual address space. Arguments are as follows:

tag This is the bus_dma_tag_t passed down from the parent driver via *<bus>_attach_args*.

segs The array of bus_dma_segment_t's filled in by **bus_dmamem_alloc**(), representing the memory regions to map.

nsegs The number of segments in *segs*.

size The size of the mapping.

kvap Filled in to specify the kernel virtual address where the memory is mapped.

flags Flags are defined as follows:

BUS_DMA_WAITOK It is safe to wait (sleep) for resources during this call.

BUS_DMA_NOWAIT It is not safe to wait (sleep) for resources during this call.

BUS_DMA_BUS[1-4] These flags are placeholders, and may be used by busses to provide bus-dependent functionality.

BUS_DMA_COHERENT This flag is a *hint* to machine-dependent code. If possible, map the memory in such a way as it will be DMA coherent. This may include mapping the pages into uncached address space or setting the cache-inhibit bits in page table entries. If DMA coherent mappings are impossible, this flag is silently ignored.

Later, when this memory is loaded into a DMA map, machine-dependent code will take whatever steps are necessary to determine if the memory was mapped in a DMA coherent fashion. This may include checking if the kernel virtual address lies within uncached address space or if the cache-inhibit bits are set in page table entries. If it is determined that the mapping is DMA coherent, state may be placed into the DMA map for use by later calls to **bus_dmamap_sync**().

Note that a device driver must not rely on BUS_DMA_COHERENT for correct operation. All calls to **bus_dmamap_sync**() must still be made. This flag is provided only as an optimization hint to machine-dependent code.

Also note that this flag only applies to coherency between the CPU and memory. Coherency between memory and the device is controlled with a different flag. See the description of the **bus_dmamap_load**() function.

BUS_DMA_NOCACHE This flag is a *hint* to machine-dependent code. If possible, map the uncached memory. This flag may be useful in the case that the memory cache causes unexpected behavior of the device.

Behavior is undefined if invalid arguments are passed to **bus_dmamem_map**().

Returns 0 on success, or an error code indicating mode of failure.

bus_dmamem_unmap(*tag*, *kva*, *size*)

Unmaps memory previously mapped with **bus_dmamem_map**(), freeing the kernel virtual address space used by the mapping. The arguments are as follows:

tag This is the bus_dma_tag_t passed down from the parent driver via *<bus>_attach_args*.

kva The kernel virtual address of the mapped memory.

size The size of the mapping.

Behavior is undefined if invalid arguments are passed to **bus_dmamem_unmap**().

If given valid arguments, **bus_dmamem_unmap**() always succeeds.

bus_dmamem_mmap(*tag*, *segs*, *nsegs*, *off*, *prot*, *flags*)

Provides support for user mmap(2)'ing of DMA-safe memory. This function is to be called by a device driver's (*d_mmap)() entry point, which is called by the device pager for each page to be mapped. The arguments are as follows:

tag This is the bus_dma_tag_t passed down from the parent driver via *<bus>_attach_args*.

segs The array of bus_dma_segment_t's filled in by **bus_dmamem_alloc**(), representing the memory to be mmap(2)'ed.

nsegs The number of elements in the *segs* array.

off The offset of the page in DMA memory which is to be mapped.

prot The protection codes for the mapping.

flags Flags are defined as follows:

BUS_DMA_WAITOK	It is safe to wait (sleep) for resources during this call.
BUS_DMA_NOWAIT	It is not safe to wait (sleep) for resources during this call.
BUS_DMA_BUS[1-4]	These flags are placeholders, and may be used by busses to provide bus-dependent functionality.
BUS_DMA_COHERENT	See **bus_dmamem_map**() above for a description of this flag.
BUS_DMA_NOCACHE	See **bus_dmamem_map**() above for a description of this flag.

Behavior is undefined if invalid arguments are passed to **bus_dmamem_mmap**().

Returns -1 to indicate failure. Otherwise, returns an opaque value to be interpreted by the device pager.

bus_dmatag_subregion(*tag*, *min_addr*, *max_addr*, *newtag*, *flags*)

Given a bus_dma_tag_t create a new bus_dma_tag_t with a limited bus address space. This function should not normally be used, but is useful for devices that do not support the full address space of the parent bus. The arguments are as follows:

tag This is the bus_dma_tag_t to subregion.

min_addr The smallest address this new tag can address.

max_addr.
The largest address this new tag can address.

newtag Pointer filled in with the address of the new bus_dma_tag_t.

flags Flags are defined as follows:

BUS_DMA_WAITOK	It is safe to wait (sleep) for resources during this call.
BUS_DMA_NOWAIT	It is not safe to wait (sleep) for resources during this call.

bus_dmatag_destroy(*tag*)

Free a tag created by **bus_dmatag_subregion**().

SEE ALSO

bus_space(9), mb(9)

Jason Thorpe, "A Machine-Independent DMA Framework for NetBSD", *Proceedings of the FREENIX track: 1998 USENIX Annual Technical Conference*, pp. 1-12, 1998.

HISTORY

The **bus_dma** interface appeared in NetBSD 1.3.

AUTHORS

The **bus_dma** interface was designed and implemented by Jason R. Thorpe of the Numerical Aerospace Simulation Facility, NASA Ames Research Center. Additional input on the **bus_dma** design was provided by Chris Demetriou, Charles Hannum, Ross Harvey, Matthew Jacob, Jonathan Stone, and Matt Thomas.

NAME

bus_space, **bus_space_barrier**, **bus_space_copy_region_1**, **bus_space_copy_region_2**, **bus_space_copy_region_4**, **bus_space_copy_region_8**, **bus_space_free**, **bus_space_map**, **bus_space_peek_1**, **bus_space_peek_2**, **bus_space_peek_4**, **bus_space_peek_8**, **bus_space_poke_1**, **bus_space_poke_2**, **bus_space_poke_4**, **bus_space_poke_8**, **bus_space_read_1**, **bus_space_read_2**, **bus_space_read_4**, **bus_space_read_8**, **bus_space_read_multi_1**, **bus_space_read_multi_2**, **bus_space_read_multi_4**, **bus_space_read_multi_8**, **bus_space_read_multi_stream_1**, **bus_space_read_multi_stream_2**, **bus_space_read_multi_stream_4**, **bus_space_read_multi_stream_8**, **bus_space_read_region_1**, **bus_space_read_region_2**, **bus_space_read_region_4**, **bus_space_read_region_8**, **bus_space_read_region_stream_1**, **bus_space_read_region_stream_2**, **bus_space_read_region_stream_4**, **bus_space_read_region_stream_8**, **bus_space_read_stream_1**, **bus_space_read_stream_2**, **bus_space_read_stream_4**, **bus_space_read_stream_8**, **bus_space_set_region_1**, **bus_space_set_region_2**, **bus_space_set_region_4**, **bus_space_set_region_8**, **bus_space_subregion**, **bus_space_unmap**, **bus_space_vaddr**, **bus_space_mmap**, **bus_space_write_1**, **bus_space_write_2**, **bus_space_write_4**, **bus_space_write_8**, **bus_space_write_multi_1**, **bus_space_write_multi_2**, **bus_space_write_multi_4**, **bus_space_write_multi_8**, **bus_space_write_multi_stream_1**, **bus_space_write_multi_stream_2**, **bus_space_write_multi_stream_4**, **bus_space_write_multi_stream_8**, **bus_space_write_region_1**, **bus_space_write_region_2**, **bus_space_write_region_4**, **bus_space_write_region_8**, **bus_space_write_region_stream_1**, **bus_space_write_region_stream_2**, **bus_space_write_region_stream_4**, **bus_space_write_region_stream_8**, **bus_space_write_stream_1**, **bus_space_write_stream_2**, **bus_space_write_stream_4**, **bus_space_write_stream_8** — bus space manipulation functions

SYNOPSIS

```
#include <machine/bus.h>

int
bus_space_map(bus_space_tag_t space, bus_addr_t address, bus_size_t size,
    int flags, bus_space_handle_t *handlep);

void
bus_space_unmap(bus_space_tag_t space, bus_space_handle_t handle,
    bus_size_t size);

int
bus_space_subregion(bus_space_tag_t space, bus_space_handle_t handle,
    bus_size_t offset, bus_size_t size, bus_space_handle_t *nhandlep);

int
bus_space_alloc(bus_space_tag_t space,bus_addr_t reg_start,
    bus_addr_t reg_end,bus_size_t size,bus_size_t alignment,
    bus_size_t boundary,int flags,bus_addr_t *addrp,
    bus_space_handle_t *handlep);

void
bus_space_free(bus_space_tag_t space, bus_space_handle_t handle,
    bus_size_t size);
```

*void **
bus_space_vaddr(*bus_space_tag_t space*, *bus_space_handle_t handle*);

paddr_t
bus_space_mmap(*bus_space_tag_t space*, *bus_addr_t addr*, *off_t off*, *int prot*, *int flags*);

int
bus_space_peek_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*);

int
bus_space_peek_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*);

int
bus_space_peek_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t *datap*);

int
bus_space_peek_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*);

int
bus_space_poke_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t data*);

int
bus_space_poke_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t data*);

int
bus_space_poke_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t data*);

int
bus_space_poke_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t data*);

uint8_t
bus_space_read_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

uint16_t
bus_space_read_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

uint32_t
bus_space_read_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

uint64_t
bus_space_read_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

void
bus_space_write_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t value*);

void
bus_space_write_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t value*);

void
bus_space_write_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t value*);

void
bus_space_write_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t value*);

void
bus_space_barrier(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *bus_size_t length*, *int flags*);

void
bus_space_read_region_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*, *bus_size_t count*);

void
bus_space_read_region_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_read_region_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t *datap*, *bus_size_t count*);

void
bus_space_read_region_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*, *bus_size_t count*);

void
bus_space_read_region_stream_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*, *bus_size_t count*);

void
bus_space_read_region_stream_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_read_region_stream_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t *datap*, *bus_size_t count*);

void
bus_space_read_region_stream_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*, *bus_size_t count*);

void
bus_space_write_region_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *const uint8_t *datap*, *bus_size_t count*);

void
bus_space_write_region_2(*bus_space_tag_t space*, *bus_space_handle_t handle*,

bus_size_t offset, *const uint16_t *datap*, *bus_size_t count*);

void
bus_space_write_region_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *const uint32_t *datap*, *bus_size_t count*);

void
bus_space_write_region_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *const uint64_t *datap*, *bus_size_t count*);

void
bus_space_write_region_stream_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *const uint8_t *datap*, *bus_size_t count*);

void
bus_space_write_region_stream_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *const uint16_t *datap*, *bus_size_t count*);

void
bus_space_write_region_stream_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *const uint32_t *datap*, *bus_size_t count*);

void
bus_space_write_region_stream_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *const uint64_t *datap*, *bus_size_t count*);

void
bus_space_copy_region_1(*bus_space_tag_t space*, *bus_space_handle_t srchandle*, *bus_size_t srcoffset*, *bus_space_handle_t dsthandle*, *bus_size_t dstoffset*, *bus_size_t count*);

void
bus_space_copy_region_2(*bus_space_tag_t space*, *bus_space_handle_t srchandle*, *bus_size_t srcoffset*, *bus_space_handle_t dsthandle*, *bus_size_t dstoffset*, *bus_size_t count*);

void
bus_space_copy_region_4(*bus_space_tag_t space*, *bus_space_handle_t srchandle*, *bus_size_t srcoffset*, *bus_space_handle_t dsthandle*, *bus_size_t dstoffset*, *bus_size_t count*);

void
bus_space_copy_region_8(*bus_space_tag_t space*, *bus_space_handle_t srchandle*, *bus_size_t srcoffset*, *bus_space_handle_t dsthandle*, *bus_size_t dstoffset*, *bus_size_t count*);

void
bus_space_set_region_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t value*, *bus_size_t count*);

void
bus_space_set_region_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t value*, *bus_size_t count*);

void
bus_space_set_region_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t value*, *bus_size_t count*);

void
bus_space_set_region_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t value*, *bus_size_t count*);

void
bus_space_read_multi_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_stream_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_stream_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_stream_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_stream_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *const uint8_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *const uint16_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *const uint32_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_8(*bus_space_tag_t space*, *bus_space_handle_t handle*,

bus_size_t offset, *const uint64_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_stream_1(*bus_space_tag_t space*,
bus_space_handle_t handle, *bus_size_t offset*, *const uint8_t *datap*,
bus_size_t count);

void
bus_space_write_multi_stream_2(*bus_space_tag_t space*,
bus_space_handle_t handle, *bus_size_t offset*, *const uint16_t *datap*,
bus_size_t count);

void
bus_space_write_multi_stream_4(*bus_space_tag_t space*,
bus_space_handle_t handle, *bus_size_t offset*, *const uint32_t *datap*,
bus_size_t count);

void
bus_space_write_multi_stream_8(*bus_space_tag_t space*,
bus_space_handle_t handle, *bus_size_t offset*, *const uint64_t *datap*,
bus_size_t count);

DESCRIPTION

The **bus_space** functions exist to allow device drivers machine-independent access to bus memory and register areas. All of the functions and types described in this document can be used by including the ⟨machine/bus.h⟩ header file.

Many common devices are used on multiple architectures, but are accessed differently on each because of architectural constraints. For instance, a device which is mapped in one system's I/O space may be mapped in memory space on a second system. On a third system, architectural limitations might change the way registers need to be accessed (e.g., creating a non-linear register space). In some cases, a single driver may need to access the same type of device in multiple ways in a single system or architecture. The goal of the **bus_space** functions is to allow a single driver source file to manipulate a set of devices on different system architectures, and to allow a single driver object file to manipulate a set of devices on multiple bus types on a single architecture.

Not all busses have to implement all functions described in this document, though that is encouraged if the operations are logically supported by the bus. Unimplemented functions should cause compile-time errors if possible.

All of the interface definitions described in this document are shown as function prototypes and discussed as if they were required to be functions. Implementations are encouraged to implement prototyped (type-checked) versions of these interfaces, but may implement them as macros if appropriate. Machine-dependent types, variables, and functions should be marked clearly in ⟨machine/bus.h⟩ to avoid confusion with the machine-independent types and functions, and, if possible, should be given names which make the machine-dependence clear.

CONCEPTS AND GUIDELINES

Bus spaces are described by bus space tags, which can be created only by machine-dependent code. A given machine may have several different types of bus space (e.g., memory space and I/O space), and thus may provide multiple different bus space tags. Individual busses or devices on a machine may use more than one bus space tag. For instance, ISA devices are given an ISA memory space tag and an ISA I/O space tag. Architectures may have several different tags which represent the same type of space, for instance because of multiple different host bus interface chipsets.

A range in bus space is described by a bus address and a bus size. The bus address describes the start of the range in bus space. The bus size describes the size of the range in bytes. Busses which are not byte addressable may require use of bus space ranges with appropriately aligned addresses and properly rounded sizes.

Access to regions of bus space is facilitated by use of bus space handles, which are usually created by mapping a specific range of a bus space. Handles may also be created by allocating and mapping a range of bus space, the actual location of which is picked by the implementation within bounds specified by the caller of the allocation function.

All of the bus space access functions require one bus space tag argument, at least one handle argument, and at least one offset argument (a bus size). The bus space tag specifies the space, each handle specifies a region in the space, and each offset specifies the offset into the region of the actual location(s) to be accessed. Offsets are given in bytes, though busses may impose alignment constraints. The offset used to access data relative to a given handle must be such that all of the data being accessed is in the mapped region that the handle describes. Trying to access data outside that region is an error.

Because some architectures' memory systems use buffering to improve memory and device access performance, there is a mechanism which can be used to create "barriers" in the bus space read and write stream.

There are two types of barriers: ordering barriers and completion barriers.

Ordering barriers prevent some operations from bypassing other operations. They are relatively light weight and described in terms of the operations they are intended to order. The important thing to note is that they create specific ordering constraint surrounding bus accesses but do not necessarily force any synchronization themselves. So, if there is enough distance between the memory operations being ordered, the preceding ones could complete by themselves resulting in no performance penalty.

For instance, a write before read barrier will force any writes issued before the barrier instruction to complete before any reads after the barrier are issued. This forces processors with write buffers to read data from memory rather than from the pending write in the write buffer.

Ordering barriers are usually sufficient for most circumstances, and can be combined together. For instance a read before write barrier can be combined with a write before write barrier to force all memory operations to complete before the next write is started.

Completion barriers force all memory operations and any pending exceptions to be completed before any instructions after the barrier may be issued. Completion barriers are extremely expensive and almost never required in device driver code. A single completion barrier can force the processor to stall on memory for hundreds of cycles on some machines.

Correctly-written drivers will include all appropriate barriers, and assume only the read/write ordering imposed by the barrier operations.

People trying to write portable drivers with the **bus_space** functions should try to make minimal assumptions about what the system allows. In particular, they should expect that the system requires bus space addresses being accessed to be naturally aligned (i.e., base address of handle added to offset is a multiple of the access size), and that the system does alignment checking on pointers (i.e., pointer to objects being read and written must point to properly-aligned data).

The descriptions of the **bus_space** functions given below all assume that they are called with proper arguments. If called with invalid arguments or arguments that are out of range (e.g., trying to access data outside of the region mapped when a given handle was created), undefined behaviour results. In that case, they may cause the system to halt, either intentionally (via panic) or unintentionally (by causing a fatal trap or by some other means) or may cause improper operation which is not immediately fatal. Functions which return void or which return data read from bus space (i.e., functions which don't obviously return an error code) do not fail. They could only fail if given invalid arguments, and in that case their behaviour is undefined. Functions which take a count of bytes have undefined results if the specified *count* is zero.

TYPES

Several types are defined in ⟨machine/bus.h⟩ to facilitate use of the **bus_space** functions by drivers.

bus_addr_t

The *bus_addr_t* type is used to describe bus addresses. It must be an unsigned integral type capable of holding the largest bus address usable by the architecture. This type is primarily used when mapping and unmapping bus space.

bus_size_t

The *bus_size_t* type is used to describe sizes of ranges in bus space. It must be an unsigned integral type capable of holding the size of the largest bus address range usable on the architecture. This type is used by virtually all of the **bus_space** functions, describing sizes when mapping regions and offsets into regions when performing space access operations.

bus_space_tag_t

The *bus_space_tag_t* type is used to describe a particular bus space on a machine. Its contents are machine-dependent and should be considered opaque by machine-independent code. This type is used by all **bus_space** functions to name the space on which they're operating.

bus_space_handle_t

The *bus_space_handle_t* type is used to describe a mapping of a range of bus space. Its contents are machine-dependent and should be considered opaque by machine-independent code. This type is used when performing bus space access operations.

MAPPING AND UNMAPPING BUS SPACE

Bus space must be mapped before it can be used, and should be unmapped when it is no longer needed. The **bus_space_map**() and **bus_space_unmap**() functions provide these capabilities.

Some drivers need to be able to pass a subregion of already-mapped bus space to another driver or module within a driver. The **bus_space_subregion**() function allows such subregions to be created.

bus_space_map(*space*, *address*, *size*, *flags*, *handlep*)

The **bus_space_map**() function maps the region of bus space named by the *space*, *address*, and *size* arguments. If successful, it returns zero and fills in the bus space handle pointed to by *handlep* with the handle that can be used to access the mapped region. If unsuccessful, it will return non-zero and leave the bus space handle pointed to by *handlep* in an undefined state.

The *flags* argument controls how the space is to be mapped. Supported flags include:

BUS_SPACE_MAP_CACHEABLE
: Try to map the space so that accesses can be cached by the system cache. If this flag is not specified, the implementation should map the space so that it will not be cached. This mapping method will only be useful in very rare occasions.

  This flag must have a value of 1 on all implementations for backward compatibility.

BUS_SPACE_MAP_PREFETCHABLE
: Try to map the space so that accesses can be prefetched by the system, and writes can be buffered. This means, accesses should be side effect free (idempotent). The **bus_space_barrier**() methods will flush the write buffer or force actual read accesses. If this flag is not specified, the implementation should map the space so that it will not be prefetched or delayed.

`BUS_SPACE_MAP_LINEAR` Try to map the space so that its contents can be accessed linearly via normal memory access methods (e.g., pointer dereferencing and structure accesses). The **bus_space_vaddr**() method can be used to obtain the kernel virtual address of the mapped range. This is useful when software wants to do direct access to a memory device, e.g., a frame buffer. If this flag is specified and linear mapping is not possible, the **bus_space_map**() call should fail. If this flag is not specified, the system may map the space in whatever way is most convenient. Use of this mapping method is not encouraged for normal device access; where linear access is not essential, use of the **bus_space_read/write**() methods is strongly recommended.

Not all combinations of flags make sense or are supported with all spaces. For instance, `BUS_SPACE_MAP_CACHEABLE` may be meaningless when used on many systems' I/O port spaces, and on some systems `BUS_SPACE_MAP_LINEAR` without `BUS_SPACE_MAP_PREFETCHABLE` may never work. When the system hardware or firmware provides hints as to how spaces should be mapped (e.g., the PCI memory mapping registers' "prefetchable" bit), those hints should be followed for maximum compatibility. On some systems, requesting a mapping that cannot be satisfied (e.g., requesting a non-prefetchable mapping when the system can only provide a prefetchable one) will cause the request to fail.

Some implementations may keep track of use of bus space for some or all bus spaces and refuse to allow duplicate allocations. This is encouraged for bus spaces which have no notion of slot-specific space addressing, such as ISA and VME, and for spaces which coexist with those spaces (e.g., EISA and PCI memory and I/O spaces co-existing with ISA memory and I/O spaces).

Mapped regions may contain areas for which there is no device on the bus. If space in those areas is accessed, the results are bus-dependent.

bus_space_unmap(*space*, *handle*, *size*)

The **bus_space_unmap**() function unmaps a region of bus space mapped with **bus_space_map**(). When unmapping a region, the *size* specified should be the same as the size given to **bus_space_map**() when mapping that region.

After **bus_space_unmap**() is called on a handle, that handle is no longer valid. (If copies were made of the handle they are no longer valid, either.)

This function will never fail. If it would fail (e.g., because of an argument error), that indicates a software bug which should cause a panic. In that case, **bus_space_unmap**() will never return.

bus_space_subregion(*space*, *handle*, *offset*, *size*, *nhandlep*)

The **bus_space_subregion**() function is a convenience function which makes a new handle to some subregion of an already-mapped region of bus space. The subregion described by the new handle starts at byte offset *offset* into the region described by *handle*, with the size given by *size*, and must be wholly contained within the original region.

If successful, **bus_space_subregion**() returns zero and fills in the bus space handle pointed to by *nhandlep*. If unsuccessful, it returns non-zero and leaves the bus space handle pointed to by *nhandlep* in an undefined state. In either case, the handle described by *handle* remains valid and is unmodified.

When done with a handle created by **bus_space_subregion**(), the handle should be thrown away. Under no circumstances should **bus_space_unmap**() be used on the handle. Doing so may confuse any resource management being done on the space, and will result in undefined behaviour. When **bus_space_unmap**() or **bus_space_free**() is called on a handle, all subregions of that handle become invalid.

bus_space_vaddr(*tag*, *handle*)

This method returns the kernel virtual address of a mapped bus space if and only if it was mapped with the BUS_SPACE_MAP_LINEAR flag. The range can be accessed by normal (volatile) pointer dereferences. If mapped with the BUS_SPACE_MAP_PREFETCHABLE flag, the **bus_space_barrier**() method must be used to force a particular access order.

bus_space_mmap(*tag*, *addr*, *off*, *prot*, *flags*)

This method is used to provide support for memory mapping bus space into user applications. If an address space is addressable via volatile pointer dereferences, **bus_space_mmap**() will return the physical address (possibly encoded as a machine-dependent cookie) of the bus space indicated by *addr* and *off*. *addr* is the base address of the device or device region, and *off* is the offset into that region that is being requested. If the request is made with BUS_SPACE_MAP_LINEAR as a flag, then a linear region must be returned to the caller. If the region cannot be mapped (either the address does not exist, or the constraints can not be met), **bus_space_mmap**() returns -1 to indicate failure.

Note that it is not necessary that the region being requested by a **bus_space_mmap**() call be mapped into a *bus_space_handle_t*.

bus_space_mmap() is called once per PAGE_SIZE page in the range. The *prot* argument indicates the memory protection requested by the user application for the range.

ALLOCATING AND FREEING BUS SPACE

Some devices require or allow bus space to be allocated by the operating system for device use. When the devices no longer need the space, the operating system should free it for use by other devices. The **bus_space_alloc**() and **bus_space_free**() functions provide these capabilities.

bus_space_alloc(*space*, *reg_start*, *reg_end*, *size*, *alignment*, *boundary*, *flags*, *addrp*, *handlep*)

The **bus_space_alloc**() function allocates and maps a region of bus space with the size given by *size*, corresponding to the given constraints. If successful, it returns zero, fills in the bus address pointed to by *addrp* with the bus space address of the allocated region, and fills in the bus space handle pointed to by *handlep* with the handle that can be used to access that region. If unsuccessful, it returns non-zero and leaves the bus address pointed to by *addrp* and the bus space handle pointed to by *handlep* in an undefined state.

Constraints on the allocation are given by the *reg_start*, *reg_end*, *alignment*, and *boundary* parameters. The allocated region will start at or after *reg_start* and end before or at *reg_end*. The *alignment* constraint must be a power of two, and the allocated region will start at an address that is an even multiple of that power of two. The *boundary* constraint, if non-zero, ensures that the region is allocated so that *first address in region* / *boundary* has the same value as *last address in region* / *boundary*. If the constraints cannot be met, **bus_space_alloc**() will fail. It is an error to specify a set of constraints that can never be met (for example, *size* greater than *boundary*).

The *flags* parameter is the same as the like-named parameter to *bus_space_map*, the same flag values should be used, and they have the same meanings.

Handles created by **bus_space_alloc**() should only be freed with **bus_space_free**(). Trying to use **bus_space_unmap**() on them causes undefined behaviour. The **bus_space_subregion**() function can be used on handles created by **bus_space_alloc**().

bus_space_free(*space*, *handle*, *size*)

The **bus_space_free**() function unmaps and frees a region of bus space mapped and allocated with **bus_space_alloc**(). When unmapping a region, the *size* specified should be the same as the size given to **bus_space_alloc**() when allocating the region.

After **bus_space_free**() is called on a handle, that handle is no longer valid. (If copies were made of the handle, they are no longer valid, either.)

This function will never fail. If it would fail (e.g., because of an argument error), that indicates a software bug which should cause a panic. In that case, **bus_space_free**() will never return.

READING AND WRITING SINGLE DATA ITEMS

The simplest way to access bus space is to read or write a single data item. The **bus_space_read_N**() and **bus_space_write_N**() families of functions provide the ability to read and write 1, 2, 4, and 8 byte data items on busses which support those access sizes.

bus_space_read_1(*space*, *handle*, *offset*)
bus_space_read_2(*space*, *handle*, *offset*)
bus_space_read_4(*space*, *handle*, *offset*)
bus_space_read_8(*space*, *handle*, *offset*)

The **bus_space_read_N**() family of functions reads a 1, 2, 4, or 8 byte data item from the offset specified by *offset* into the region specified by *handle* of the bus space specified by *space*. The location being read must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data item being read. On some systems, not obeying this requirement may cause incorrect data to be read, on others it may cause a system crash.

Read operations done by the **bus_space_read_N**() functions may be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function.

These functions will never fail. If they would fail (e.g., because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

bus_space_write_1(*space*, *handle*, *offset*, *value*)
bus_space_write_2(*space*, *handle*, *offset*, *value*)
bus_space_write_4(*space*, *handle*, *offset*, *value*)
bus_space_write_8(*space*, *handle*, *offset*, *value*)

The **bus_space_write_N**() family of functions writes a 1, 2, 4, or 8 byte data item to the offset specified by *offset* into the region specified by *handle* of the bus space specified by *space*. The location being written must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data item being written. On some systems, not obeying this requirement may cause incorrect data to be written, on others it may cause a system crash.

Write operations done by the **bus_space_write_N**() functions may be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function.

These functions will never fail. If they would fail (e.g., because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

PROBING BUS SPACE FOR HARDWARE WHICH MAY NOT RESPOND

One problem with the **bus_space_read_N**() and **bus_space_write_N**() family of functions is that they provide no protection against exceptions which can occur when no physical hardware or device responds to the read or write cycles. In such a situation, the system typically would panic due to a kernel-mode bus error. The **bus_space_peek_N**() and **bus_space_poke_N**() family of functions provide a mechanism to handle these exceptions gracefully without the risk of crashing the system.

As with **bus_space_read_N**() and **bus_space_write_N**(), the peek and poke functions provide the ability to read and write 1, 2, 4, and 8 byte data items on busses which support those access sizes. All of the constraints specified in the descriptions of the **bus_space_read_N**() and **bus_space_write_N**() functions also apply to **bus_space_peek_N**() and **bus_space_poke_N**().

In addition, explicit calls to the **bus_space_barrier**() function are not required as the implementation will ensure all pending operations complete before the peek or poke operation starts. The implementation will also ensure that the peek or poke operations complete before returning.

The return value indicates the outcome of the peek or poke operation. A return value of zero implies that a hardware device is responding to the operation at the specified offset in the bus space. A non-zero return value indicates that the kernel intercepted a hardware exception (e.g., bus error) when the peek or poke operation was attempted. Note that some busses are incapable of generating exceptions when non-existent hardware is accessed. In such cases, these functions will always return zero and the value of the data read by **bus_space_peek_N**() will be unspecified.

Finally, it should be noted that at this time the **bus_space_peek_N**() and **bus_space_poke_N**() functions are not re-entrant and should not, therefore, be used from within an interrupt service routine. This constraint may be removed at some point in the future.

bus_space_peek_1(*space*, *handle*, *offset*, *datap*)
bus_space_peek_2(*space*, *handle*, *offset*, *datap*)
bus_space_peek_4(*space*, *handle*, *offset*, *datap*)
bus_space_peek_8(*space*, *handle*, *offset*, *datap*)

The **bus_space_peek_N**() family of functions cautiously read a 1, 2, 4, or 8 byte data item from the offset specified by *offset* in the region specified by *handle* of the bus space specified by *space*. The data item read is stored in the location pointed to by *datap*. It is permissible for *datap* to be NULL, in which case the data item will be discarded after being read.

bus_space_poke_1(*space*, *handle*, *offset*, *value*)
bus_space_poke_2(*space*, *handle*, *offset*, *value*)
bus_space_poke_4(*space*, *handle*, *offset*, *value*)
bus_space_poke_8(*space*, *handle*, *offset*, *value*)

The **bus_space_poke_N**() family of functions cautiously write a 1, 2, 4, or 8 byte data item specified by *value* to the offset specified by *offset* in the region specified by *handle* of the bus space specified by *space*.

BARRIERS

In order to allow high-performance buffering implementations to avoid bus activity on every operation, read and write ordering should be specified explicitly by drivers when necessary. The **bus_space_barrier**() function provides that ability.

bus_space_barrier(*space*, *handle*, *offset*, *length*, *flags*)

The **bus_space_barrier**() function enforces ordering of bus space read and write operations for the specified subregion (described by the *offset* and *length* parameters) of the region named by *handle* in the space named by *space*.

The *flags* argument controls what types of operations are to be ordered. Supported flags are:

BUS_SPACE_BARRIER_READ_BEFORE_READ — Force all reads before the barrier to complete before any reads after the barrier may be issued.

`BUS_SPACE_BARRIER_READ_BEFORE_WRITE`	Force all reads before the barrier to complete before any writes after the barrier may be issued.
`BUS_SPACE_BARRIER_WRITE_BEFORE_READ`	Force all writes before the barrier to complete before any reads after the barrier may be issued.
`BUS_SPACE_BARRIER_WRITE_BEFORE_WRITE`	Force all writes before the barrier to complete before any writes after the barrier may be issued.
`BUS_SPACE_BARRIER_SYNC`	Force all memory operations and any pending exceptions to be completed before any instructions after the barrier may be issued.

Those flags can be combined (or-ed together) to enforce ordering on different combinations of read and write operations.

All of the specified type(s) of operation which are done to the region before the barrier operation are guaranteed to complete before any of the specified type(s) of operation done after the barrier.

Example: Consider a hypothetical device with two single-byte ports, one write-only input port (at offset 0) and a read-only output port (at offset 1). Operation of the device is as follows: data bytes are written to the input port, and are placed by the device on a stack, the top of which is read by reading from the output port. The sequence to correctly write two data bytes to the device then read those two data bytes back would be:

```
/*
 * t and h are the tag and handle for the mapped device's
 * space.
 */
bus_space_write_1(t, h, 0, data0);
bus_space_barrier(t, h, 0, 1, BUS_SPACE_BARRIER_WRITE_BEFORE_WRITE); /* 1 */
bus_space_write_1(t, h, 0, data1);
bus_space_barrier(t, h, 0, 2, BUS_SPACE_BARRIER_WRITE_BEFORE_READ);  /* 2 */
ndata1 = bus_space_read_1(t, h, 1);
bus_space_barrier(t, h, 1, 1, BUS_SPACE_BARRIER_READ_BEFORE_READ);   /* 3 */
ndata0 = bus_space_read_1(t, h, 1);
/* data0 == ndata0, data1 == ndata1 */
```

The first barrier makes sure that the first write finishes before the second write is issued, so that two writes to the input port are done in order and are not collapsed into a single write. This ensures that the data bytes are written to the device correctly and in order.

The second barrier forces the writes to the output port finish before any of the reads to the input port are issued, thereby making sure that all of the writes are finished before data is read. This ensures that the first byte read from the device really is the last one that was written.

The third barrier makes sure that the first read finishes before the second read is issued, ensuring that data is read correctly and in order.

The barriers in the example above are specified to cover the absolute minimum number of bus space locations. It is correct (and often easier) to make barrier operations cover the device's whole range of bus space, that is, to specify an offset of zero and the size of the whole region.

The following barrier operations are obsolete and should be removed from existing code:

`BUS_SPACE_BARRIER_READ` Synchronize read operations.

`BUS_SPACE_BARRIER_WRITE` Synchronize write operations.

REGION OPERATIONS

Some devices use buffers which are mapped as regions in bus space. Often, drivers want to copy the contents of those buffers to or from memory, e.g., into mbufs which can be passed to higher levels of the system or from mbufs to be output to a network. In order to allow drivers to do this as efficiently as possible, the **bus_space_read_region_N**() and **bus_space_write_region_N**() families of functions are provided.

Drivers occasionally need to copy one region of a bus space to another, or to set all locations in a region of bus space to contain a single value. The **bus_space_copy_region_N**() family of functions and the **bus_space_set_region_N**() family of functions allow drivers to perform these operations.

bus_space_read_region_1(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_region_2(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_region_4(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_region_8(*space*, *handle*, *offset*, *datap*, *count*)

The **bus_space_read_region_N**() family of functions reads *count* 1, 2, 4, or 8 byte data items from bus space starting at byte offset *offset* in the region specified by *handle* of the bus space specified by *space* and writes them into the array specified by *datap*. Each successive data item is read from an offset 1, 2, 4, or 8 bytes after the previous data item (depending on which function is used). All locations being read must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data items being read and the data array pointer should be properly aligned. On some systems, not obeying these requirements may cause incorrect data to be read, on others it may cause a system crash.

Read operations done by the **bus_space_read_region_N**() functions may be executed in any order. They may also be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. There is no way to insert barriers between reads of individual bus space locations executed by the **bus_space_read_region_N**() functions.

These functions will never fail. If they would fail (e.g., because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

bus_space_write_region_1(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_region_2(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_region_4(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_region_8(*space*, *handle*, *offset*, *datap*, *count*)

The **bus_space_write_region_N**() family of functions reads *count* 1, 2, 4, or 8 byte data items from the array specified by *datap* and writes them to bus space starting at byte offset *offset* in the region specified by *handle* of the bus space specified by *space*. Each successive data item is written to an offset 1, 2, 4, or 8 bytes after the previous data item (depending on which function is used). All locations being written must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data items being written and the data array pointer should be properly aligned. On some systems, not obeying these requirements may cause incorrect data to be written, on others it may cause a system crash.

Write operations done by the **bus_space_write_region_N**() functions may be executed in any order. They may also be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. There is no way to insert barriers between writes of individual bus space locations executed by the **bus_space_write_region_N**() functions.

These functions will never fail. If they would fail (e.g., because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

bus_space_copy_region_1(*space*, *srchandle*, *srcoffset*, *dsthandle*, *dstoffset*, *count*)
bus_space_copy_region_2(*space*, *srchandle*, *srcoffset*, *dsthandle*, *dstoffset*, *count*)
bus_space_copy_region_4(*space*, *srchandle*, *srcoffset*, *dsthandle*, *dstoffset*, *count*)
bus_space_copy_region_8(*space*, *srchandle*, *srcoffset*, *dsthandle*, *dstoffset*, *count*)

The **bus_space_copy_region_N**() family of functions copies *count* 1, 2, 4, or 8 byte data items in bus space from the area starting at byte offset *srcoffset* in the region specified by *srchandle* of the bus space specified by *space* to the area starting at byte offset *dstoffset* in the region specified by *dsthandle* in the same bus space. Each successive data item read or written has an offset 1, 2, 4, or 8 bytes after the previous data item (depending on which function is used). All locations being read and written must lie within the bus space region specified by their respective handles.

For portability, the starting addresses of the regions specified by each handle plus its respective offset should be a multiple of the size of data items being copied. On some systems, not obeying this requirement may cause incorrect data to be copied, on others it may cause a system crash.

Read and write operations done by the **bus_space_copy_region_N**() functions may be executed in any order. They may also be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**(*function*). There is no way to insert barriers between reads or writes of individual bus space locations executed by the **bus_space_copy_region_N**() functions.

Overlapping copies between different subregions of a single region of bus space are handled correctly by the **bus_space_copy_region_N**() functions.

These functions will never fail. If they would fail (e.g., because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

bus_space_set_region_1(*space*, *handle*, *offset*, *value*, *count*)
bus_space_set_region_2(*space*, *handle*, *offset*, *value*, *count*)
bus_space_set_region_4(*space*, *handle*, *offset*, *value*, *count*)
bus_space_set_region_8(*space*, *handle*, *offset*, *value*, *count*)

The **bus_space_set_region_N**() family of functions writes the given *value* to *count* 1, 2, 4, or 8 byte data items in bus space starting at byte offset *offset* in the region specified by *handle* of the bus space specified by *space*. Each successive data item has an offset 1, 2, 4, or 8 bytes after the previous data item (depending on which function is used). All locations being written must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data items being written. On some systems, not obeying this requirement may cause incorrect data to be written, on others it may cause a system crash.

Write operations done by the **bus_space_set_region_N**() functions may be executed in any order. They may also be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. There is no way to insert barriers between writes of individual bus space locations executed by the **bus_space_set_region_N**() functions.

These functions will never fail. If they would fail (e.g., because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

READING AND WRITING A SINGLE LOCATION MULTIPLE TIMES

Some devices implement single locations in bus space which are to be read or written multiple times to communicate data, e.g., some ethernet devices' packet buffer FIFOs. In order to allow drivers to manipulate these types of devices as efficiently as possible, the **bus_space_read_multi_N**() and **bus_space_write_multi_N**() families of functions are provided.

bus_space_read_multi_1(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_multi_2(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_multi_4(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_multi_8(*space*, *handle*, *offset*, *datap*, *count*)

The **bus_space_read_multi_N**() family of functions reads *count* 1, 2, 4, or 8 byte data items from bus space at byte offset *offset* in the region specified by *handle* of the bus space specified by *space* and writes them into the array specified by *datap*. Each successive data item is read from the same location in bus space. The location being read must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data items being read and the data array pointer should be properly aligned. On some systems, not obeying these requirements may cause incorrect data to be read, on others it may cause a system crash.

Read operations done by the **bus_space_read_multi_N**() functions may be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. Because the **bus_space_read_multi_N**() functions read the same bus space location multiple times, they place an implicit read barrier between each successive read of that bus space location.

These functions will never fail. If they would fail (e.g., because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

bus_space_write_multi_1(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_multi_2(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_multi_4(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_multi_8(*space*, *handle*, *offset*, *datap*, *count*)

The **bus_space_write_multi_N**() family of functions reads *count* 1, 2, 4, or 8 byte data items from the array specified by *datap* and writes them into bus space at byte offset *offset* in the region specified by *handle* of the bus space specified by *space*. Each successive data item is written to the same location in bus space. The location being written must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data items being written and the data array pointer should be properly aligned. On some systems, not obeying these requirements may cause incorrect data to be written, on others it may cause a system crash.

Write operations done by the **bus_space_write_multi_N**() functions may be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. Because the **bus_space_write_multi_N**() functions write the same bus space location multiple times, they place an implicit write barrier between each successive write of that bus space location.

These functions will never fail. If they would fail (e.g., because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

STREAM FUNCTIONS

Most of the **bus_space** functions imply a host byte-order and a bus byte-order and take care of any translation for the caller. In some cases, however, hardware may map a FIFO or some other memory region for which the caller may want to use multi-word, yet untranslated access. Access to these types of memory regions should be with the **bus_space_*_stream_N**() functions.

bus_space_read_stream_1(*space*, *handle*, *offset*)
bus_space_read_stream_2(*space*, *handle*, *offset*)
bus_space_read_stream_4(*space*, *handle*, *offset*)
bus_space_read_stream_8(*space*, *handle*, *offset*)
bus_space_read_multi_stream_1(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_multi_stream_2(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_multi_stream_4(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_multi_stream_8(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_region_stream_1(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_region_stream_2(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_region_stream_4(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_read_region_stream_8(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_stream_1(*space*, *handle*, *offset*, *value*)
bus_space_write_stream_2(*space*, *handle*, *offset*, *value*)
bus_space_write_stream_4(*space*, *handle*, *offset*, *value*)
bus_space_write_stream_8(*space*, *handle*, *offset*, *value*)
bus_space_write_multi_stream_1(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_multi_stream_2(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_multi_stream_4(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_multi_stream_8(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_region_stream_1(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_region_stream_2(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_region_stream_4(*space*, *handle*, *offset*, *datap*, *count*)
bus_space_write_region_stream_8(*space*, *handle*, *offset*, *datap*, *count*)

These functions are defined just as their non-stream counterparts, except that they provide no byte-order translation.

EXPECTED CHANGES TO THE BUS_SPACE FUNCTIONS

The definition of the **bus_space** functions should not yet be considered finalized. There are several changes and improvements which should be explored, including:

- Providing a mechanism by which incorrectly-written drivers will be automatically given barriers and properly-written drivers won't be forced to use more barriers than they need. This should probably be done via a `#define` in the incorrectly-written drivers. Unfortunately, at this time, few drivers actually use barriers correctly (or at all). Because of that, **bus_space** implementations on architectures which do buffering must always do the barriers inside the **bus_space** calls, to be safe. That has a potentially significant performance impact.
- Exporting the **bus_space** functions to user-land so that applications (such as X servers) have easier, more portable access to device space.
- Redefining bus space tags and handles so that machine-independent bus interface drivers (for example PCI to VME bridges) could define and implement bus spaces without requiring machine-dependent code. If this is done, it should be done in such a way that machine-dependent optimizations should remain possible.
- Converting bus spaces (such as PCI configuration space) which currently use space-specific access methods to use the **bus_space** functions where that is appropriate.
- Redefining the way bus space is mapped and allocated, so that mapping and allocation are done with bus specific functions which return bus space tags. This would allow further optimization than is currently possible, and would also ease translation of the **bus_space** functions into user space (since mapping in user space would look like it just used a different bus-specific mapping function).

COMPATIBILITY

The current version of the **bus_space** interface specification differs slightly from the original specification that came into wide use. A few of the function names and arguments have changed for consistency and increased functionality. Drivers that were written to the old, deprecated specification can be compiled by defining the `__BUS_SPACE_COMPAT_OLDDEFS` preprocessor symbol before including ⟨`machine/bus.h`⟩.

SEE ALSO

`bus_dma`(9), `mb`(9)

HISTORY

The **bus_space** functions were introduced in a different form (memory and I/O spaces were accessed via different sets of functions) in NetBSD 1.2. The functions were merged to work on generic "spaces" early in the NetBSD 1.3 development cycle, and many drivers were converted to use them. This document was written later during the NetBSD 1.3 development cycle and the specification was updated to fix some consistency problems and to add some missing functionality.

AUTHORS

The **bus_space** interfaces were designed and implemented by the NetBSD developer community. Primary contributors and implementors were Chris Demetriou, Jason Thorpe, and Charles Hannum, but the rest of the NetBSD developers and the user community played a significant role in development.

Chris Demetriou wrote this manual page.

NAME

bzero — write zeroes to a byte string

SYNOPSIS

```
#include <sys/systm.h>

void
bzero(void *b, size_t len);
```

DESCRIPTION

The bzero() interface is obsolete. Do not add new code using it. It will soon be purged. Use memset(9) instead. (The bzero() function is now a macro for memset(9).)

The **bzero**() function writes *len* zero bytes to the string *b*. If *len* is zero, **bzero**() does nothing.

SEE ALSO

memset(9)

NAME

callout_init, **callout_destroy**, **callout_reset**, **callout_schedule**, **callout_setfunc**, **callout_stop**, **callout_expired**, **callout_invoking**, **callout_ack** — execute a function after a specified length of time

SYNOPSIS

```
#include <sys/callout.h>

void
callout_init(callout_t *c, u_int flags);

void
callout_destroy(callout_t *c);

void
callout_reset(callout_t *c, int ticks, void (*func)(void *), void *arg);

void
callout_schedule(callout_t *c, int ticks);

void
callout_setfunc(callout_t *c, void (*func)(void *), void *arg);

bool
callout_stop(callout_t *c);

bool
callout_pending(callout_t *c);

bool
callout_expired(callout_t *c);

bool
callout_active(callout_t *c);

bool
callout_invoking(callout_t *c);

bool
callout_ack(callout_t *c);
```

DESCRIPTION

The **callout** facility provides a mechanism to execute a function at a given time. The timer is based on the hardclock timer which ticks hz times per second. The function is called at softclock interrupt level.

Clients of the **callout** facility are responsible for providing pre-allocated callout structures, or "handles". The **callout** facility replaces the historic UNIX functions **timeout**() and **untimeout**().

The **callout_init**() function initializes the callout handle *c* for use. No operations can be performed on the callout before it is initialized. If the *flags* argument is CALLOUT_MPSAFE, the handler will be called without getting the global kernel lock. In this case it should only use functions that are multiprocessor safe.

callout_destroy() destroys the callout, preventing further use. It is provided as a diagnostic facility intended to catch bugs. To ensure future compatibility, **callout_destroy**() should always be called when the callout is no longer required (for instance, when a device is being detached).

The **callout_reset**() function resets and starts the timer associated with the callout handle *c*. When the timer expires after *ticks*/hz seconds, the function specified by *func* will be called with the argument *arg*. If the timer associated with the callout handle is already running, the callout will simply be rescheduled to

execute at the newly specified time. Once the timer is started, the callout handle is marked as *PENDING*. Once the timer expires, the handle is marked as *EXPIRED* and *INVOKING*, and the *PENDING* status is cleared.

The **callout_setfunc**() function sets the function and argument of the callout handle *c* to *func* and *arg* respectively. The callout handle must already be initialized. If a callout will always be used with the same function and argument, then **callout_setfunc**() used in conjunction with **callout_schedule**() is slightly more efficient than using **callout_reset**().

The **callout_stop**() function stops the timer associated the callout handle *c*. The *PENDING* and *EXPIRED* status for the callout handle is cleared. It is safe to call **callout_stop**() on a callout handle that is not pending, so long as it is initialized. **callout_stop**() will return a non-zero value if the callout was *EXPIRED*.

The **callout_pending**() function tests the *PENDING* status of the callout handle *c*. A *PENDING* callout is one that has been started and whose function has not yet been called. Note that it is possible for a callout's timer to have expired without its function being called if interrupt level has not dropped low enough to let softclock interrupts through. Note that it is only safe to test *PENDING* status when at softclock interrupt level or higher.

The **callout_expired**() function tests to see if the callout's timer has expired and its function called.

The **callout_active**() function returns true if a timer has been started but not explicitly stopped, even if it has already fired. **callout_active**(*foo*) is logically the same as **callout_pending**(*foo*) || **callout_expired**(*foo*); it is implemented as a separate function for compatibility with FreeBSD and for the special case of **TCP_TIMER_ISARMED**(). Its use is not recommended.

The **callout_invoking**() function tests the *INVOKING* status of the callout handle *c*. This flag is set just before a callout's function is being called. Since the priority level is lowered prior to invocation of the callout function, other pending higher-priority code may run before the callout function is allowed to run. This may create a race condition if this higher-priority code deallocates storage containing one or more callout structures whose callout functions are about to be run. In such cases, one technique to prevent references to deallocated storage would be to test whether any callout functions are in the *INVOKING* state using **callout_invoking**(), and if so, to mark the data structure and defer storage deallocation until the callout function is allowed to run. For this handshake protocol to work, the callout function will have to use the **callout_ack**() function to clear this flag.

The **callout_ack**() function clears the *INVOKING* state in the callout handle *c*. This is used in situations where it is necessary to protect against the race condition described under **callout_invoking**().

SEE ALSO

hz(9)

HISTORY

The **callout** facility was implemented by Artur Grabowski and Thomas Nordin, based on the work of G. Varghese and A. Lauck, described in the paper Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility in the Proceedings of the 11th ACM Annual Symposium on Operating System Principles, Austin, Texas, November 1987. It was adapted to the NetBSD kernel by Jason R. Thorpe.

NAME

Cardbus, **cardbus_attach_card**, **cardbus_detach_card**, **cardbus_function_enable**, **cardbus_function_disable**, **cardbus_mapreg_map**, **cardbus_mapreg_unmap**, **cardbus_get_capability**, **cardbus_make_tag**, **cardbus_free_tag**, **cardbus_conf_read**, **cardbus_conf_write**, **cardbus_intr_establish**, **cardbus_intr_disestablish**, **CARDBUS_VENDOR**, **CARDBUS_PRODUCT**, **Cardbus_function_enable**, **Cardbus_function_disable**, **Cardbus_mapreg_map**, **Cardbus_mapreg_unmap**, **Cardbus_make_tag**, **Cardbus_free_tag**, **Cardbus_conf_read**, **Cardbus_conf_write** — support for CardBus PC-Card devices

SYNOPSIS

```
#include <machine/bus.h>
#include <dev/cardbus/cardbusvar.h>
#include <dev/cardbus/cardbusreg.h>
#include <dev/cardbus/cardbusdevs.h>

int
cardbus_attach_card(struct cardbus_softc *csc);

void
cardbus_detach_card(struct cardbus_softc *csc);

int
cardbus_function_enable(struct cardbus_softc *csc, int function);

int
cardbus_function_disable(struct cardbus_softc *csc, int function);

int
cardbus_mapreg_map(struct cardbus_softc *csc,int cf,int reg,
    cardbusreg_t type,int busflags,bus_space_tag_t *tagp,
    bus_space_handle_t *handlep,bus_addr_t *basep,bus_size_t *sizep);

int
cardbus_mapreg_unmap(struct cardbus_softc *csc, int cf, int reg,
    bus_space_tag_t tag, bus_space_handle_t handle, bus_size_t size);

int
cardbus_get_capability(cardbus_chipset_tag_t cc,
    cardbus_function_tag_t cf, cardbustag_t tag, int capid, int *offsetp,
    cardbusreg_t *valuep);

cardbustag_t
cardbus_make_tag(cardbus_chipset_tag_t cc, int cf, int bus, int device,
    int function);

void
cardbus_free_tag(cardbus_chipset_tag_t cc, int cf, cardbustag_t tag);

cardbusreg_t
cardbus_conf_read(cardbus_chipset_tag_t cc, int cf, cardbustag_t tag,
    int offs);

void
cardbus_conf_write(cardbus_chipset_tag_t cc, int cf, cardbustag_t tag,
    int offs, busreg_t val);
```

*void **
cardbus_intr_establish(*cardbus_chipset_tag_t cc*,
cardbus_function_tag_t cf, *cardbus_intr_handle_t irq*, *int level*,
*int (*handler)(void *)*, *void *arg*);

void
cardbus_intr_disestablish(*cardbus_chipset_tag_t cc*,
cardbus_function_tag_t cf, *void *ih*);

int
CARDBUS_VENDOR(*cardbusreg_t id*);

int
CARDBUS_PRODUCT(*cardbusreg_t id*);

int
Cardbus_function_enable(*cardbus_devfunc_t ct*);

int
Cardbus_function_disable(*cardbus_devfunc_t ct*);

int
Cardbus_mapreg_map(*cardbus_devfunc_t ct*, *int reg*, *cardbusreg_t type*,
int busflags, *bus_space_tag_t *tagp*, *bus_space_handle_t *handlep*,
*bus_addr_t *basep*, *bus_size_t *sizep*);

int
Cardbus_mapreg_unmap(*cardbus_devfunc_t ct*, *int reg*, *bus_space_tag_t tag*,
bus_space_handle_t handle, *bus_size_t size*);

cardbustag_t
Cardbus_make_tag(*cardbus_devfunc_t ct*);

void
Cardbus_free_tag(*cardbus_devfunc_t ct*, *cardbustag_t tag*);

cardbusreg_t
Cardbus_conf_read(*cardbus_devfunc_t ct*, *cardbustag_t tag*, *int offs*);

void
Cardbus_conf_write(*cardbus_devfunc_t ct*, *cardbustag_t tag*, *int offs*,
busreg_t val);

DESCRIPTION

The machine-independent **Cardbus** subsystem provides support for CardBus devices.

The CardBus interface is an improvement to the PC-Card interface supported by pcmcia(9). It introduces several new capabilities such as 32-bit addressing, 33MHz operation, busmaster operation and 3.3 volt low-voltage power. It remains compatible with all features of the PC-Card standard.

The CardBus interface signaling protocol is derived from the PCI signaling protocol. There are some differences between PCI and CardBus, however operations are identical for most functions implemented. Since a 32-bit CardBus interface is also defined for 16-bit PC-Cards, the same Card Services client to be used to manage both CardBus and PCMCIA PC-Cards. By interrogating the card upon detection of an insertion event, NetBSD determines whether the card requires **Cardbus** support or not, and then applies the appropriate power and signaling protocol requirements.

DATA TYPES

Drivers attached to the CardBus will make use of the following data types:

struct cardbus_attach_args
: Devices have their identity recorded in this structure. It contains the following members:

```
cardbus_devfunc_t ca_ct;
bus_space_tag_t ca_iot;       /* CardBus I/O space tag */
bus_space_tag_t ca_memt;      /* CardBus MEM space tag */
bus_dma_tag_t ca_dmat;        /* DMA tag */
u_int ca_device;
cardbustag_t ca_tag;
cardbusreg_t ca_id;
cardbusreg_t ca_class;
cardbus_intr_line_t ca_intrline; /* interrupt info */
struct cardbus_cis_info ca_cis;
```

FUNCTIONS

cardbus_attach_card(*csc*)
: Attaches the card on the slot by turning on the power, read and analyse the tuple and sets configuration index. This function returns the number of recognised device functions. If no device functions are recognised it returns zero.

cardbus_detach_card(*csc*)
: Detaches the card on the slot by release resources and turning off the power. This function must not be called under interrupt context.

cardbus_function_enable(*csc*, *function*)
: Enables device function *function* on the card. Power will be applied if it hasn't already.

cardbus_function_disable(*csc*, *function*)
: Disables device function *function* on the card. When no device functions are enabled, the turn is turned off.

cardbus_mapreg_map(*csc*, *cf*, *reg*, *type*, *busflags*, *tagp*, *handlep*, *basep*, *sizep*)
: Maps bus-space on the value of Base Address Register (BAR) indexed by *reg* for device function *cf*. The bus-space configuration is returned in *tagp*, *handlep*, *basep*, and *sizep*.

cardbus_mapreg_unmap(*csc*, *cf*, *reg*, *tag*, *handle*, *bus_size_t size*)
: Releases bus-space region for device function *cf* specified by *tag*, *handle* and *size*. *reg* is the offset of the BAR register.

cardbus_get_capability(*cc*, *cf*, *tag*, *capid*, *offsetp*, *valuep*)
: Find the PCI capability for the device function *cf* specified by *capid*. Returns the capability in *offsetp* and *valuep*.

cardbus_make_tag(*cc*, *cf*, *bus*, *device*, *function*)
: Make a tag to access config space of a CardBus card. It works the same as **pci_make_tag**().

cardbus_free_tag(*cc*, *cf*, *tag*)
: Release a tag used to access the config space of a CardBus card. It works the same as **pci_free_tag**().

cardbus_conf_read(*cc*, *cf*, *tag*, *offs*)
: Read the config space of a CardBus card. It works the same as **pci_conf_read**().

cardbus_conf_write(*cc*, *cf*, *tag*, *offs*, *val*)
Write to the config space of a CardBus card. It works same as **pci_conf_write**().

cardbus_intr_establish(*cc*, *cf*, *irq*, *level*, *handler*, *arg*)
Establish an interrupt handler for device function *cf*. The priority of the interrupt is specified by *level*. When the interrupt occurs the function *handler* is called with argument *arg*. The return value is a handle for the interrupt handler. **cardbus_intr_establish**() returns an opaque handle to an event descriptor if it succeeds, and returns NULL on failure.

cardbus_intr_disestablish(*cc*, *cf*, *ih*)
Dis-establish the interrupt handler for device function *cf* with handle *ih*. The handle was returned from **cardbus_intr_establish**().

CARDBUS_VENDOR(*id*)
Return the CardBus vendor ID for device *id*.

CARDBUS_PRODUCT(*id*)
Return the CardBus product ID for device *id*.

The **Cardbus_***() functions are convenience functions taking a *cardbus_devfunc_t* argument and perform the same operation as their namesake described above.

AUTOCONFIGURATION

During autoconfiguration, a **Cardbus** driver will receive a pointer to *struct isapnp_attach_args* describing the device attaches to the CardBus. Drivers match the device using the *ca_id* member using **CARDBUS_VENDOR**() and **CARDBUS_PRODUCT**().

During the driver attach step, drivers should initially map the device I/O and memory resources using **cardbus_mapreg_map**() or **Cardbus_mapreg_map**(). Upon successful allocation of resources, power can be applied to the device with **cardbus_function_enable**() or **Cardbus_function_enable**(). so that device-specific interrogation can be performed. Finally, power should be removed from the device using **cardbus_function_disable**() or **Cardbus_function_disable**().

Since CardBus devices support dynamic configuration, drivers should make use of **powerhook_establish**(*9*). Power can be applied and the interrupt handler should be established through this interface.

DMA SUPPORT

No additional support is provided for CardBus DMA beyond the facilities provided by the bus_dma(9) interface.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent CardBus subsystem can be found. All pathnames are relative to /usr/src.

The CardBus subsystem itself is implemented within the files sys/dev/cardbus/cardbus.c, sys/dev/cardbus/cardbus_map.c and sys/dev/cardbus/cardslot.c. The database of known devices exists within the file sys/dev/cardbus/cardbus_data.h and is generated automatically from the file sys/dev/cardbus/cardbusdevs. New vendor and product identifiers should be added to this file. The database can be regenerated using the Makefile sys/dev/cardbus/Makefile.cardbusdevs.

SEE ALSO

`cardbus`(4), `pcmcia`(4), `autoconf`(9), `bus_dma`(9), `bus_space`(9), `driver`(9), `pci`(9), `pcmcia`(9)

HISTORY

The machine-independent **`Cardbus`** subsystem appeared in NetBSD 1.5.

NAME

cn_trap, **cn_isconsole**, **cn_check_magic**, **cn_init_magic**, **cn_set_magic**, **cn_get_magic**, **cn_destroy_magic** — console magic key sequence management

SYNOPSIS

```
#include <sys/systm.h>
typedef struct cnm_state cnm_state_t;

void
cn_trap();

int
cn_isconsole(dev_t dev);

void
cn_check_magic(dev_t dev, int k, cnm_state_t *cnms);

void
cn_init_magic(cnm_state_t *cnms);

int
cn_set_magic(char *magic);

int
cn_get_magic(char *magic, int len);

void
cn_destroy_magic(cnm_state_t *cnms);
```

DESCRIPTION

The NetBSD console magic key sequence management framework is designed to provide flexible methods to set, change, and detect magic key sequences on console devices and break into the debugger or ROM monitor with a minimum of interrupt latency.

Drivers that generate console input should make use of these routines. A different *cnm_state_t* should be used for each separate input stream. Multiple devices that share the same input stream, such as USB keyboards can share the same *cnm_state_t*. Once a *cnm_state_t* is allocated, it should be initialized with **cn_init_magic**() so it can be used by **cn_check_magic**(). If a driver thinks it might be the console input device it can set the magic sequence with **cn_set_magic**() to any arbitrary string. Whenever the driver receives input, it should call **cn_check_magic**() to process the data and determine whether the magic sequence has been hit.

The magic key sequence can be accessed through the *hw.cnmagic* **sysctl** variable. This is the raw data and may be keycodes rather than processed characters, depending on the console device.

Here is a description of the console magic interface:

void cn_init_magic(*cnm_state_t *cnm*)

Initialize the console magic state pointed to by *cnm* to a usable state.

void cnm_trap()

Trap into the kernel debugger or ROM monitor. By default this routine is defined to be **console_debugger**() but can be overridden in MI header files.

int cn_isconsole(*dev_t dev*)

Determine whether a given *dev* is the system console. This macro tests to see if *dev* is the same as *cn_tab->cn_dev* but can be overridden in MI header files.

void cn_check_magic(*dev_t dev*, *int k*, *cnm_state_t *cnms*)

All input should be passed through **cn_check_magic**() so the state machine remains in a consistent state. **cn_check_magic**() calls **cn_isconsole**() with *dev* to determine if this is the console. If that returns true then it runs the input value *k* through the state machine. If the state machine completes a match of the current console magic sequence **cn_trap**() is called. Some input may need to be translated to state machine values such as the serial line BREAK sequence.

void cn_destroy_magic(*cnm_state_t *cnms*)

This should be called once what *cnms* points to is no longer needed.

int cn_set_magic(*char *magic*)

cn_set_magic() encodes a `nul` terminated string arbitrary string into values that can be used by the state machine and installs it as the global magic sequence. The escape sequence is character value `0x27` and can be used to encode special values:

0x27	The literal value `0x27`.
0x01	Serial BREAK sequence.
0x02	`Nul` character.

Returns `0` on success or a non-zero error value.

int cn_get_magic(*char *magic*, *int len*)

Extract the current magic sequence from the state machine and return up to *len* bytes of it in the buffer pointed to by *magic*. It uses the same encoding accepted by **cn_set_magic**(). Returns `0` on success or a non-zero error value.

SEE ALSO

`sysctl`(8)

HISTORY

The NetBSD console magic key sequence management framework first appeared in NetBSD 1.6.

AUTHORS

The NetBSD console magic key sequence management framework was designed and implemented by Eduardo Horvath ⟨eeh@NetBSD.org⟩.

NAME

cv, **condvar**, **cv_init**, **cv_destroy**, **cv_wait**, **cv_wait_sig**, **cv_timedwait**, **cv_timedwait_sig**, **cv_signal**, **cv_broadcast**, **cv_has_waiters** — condition variables

SYNOPSIS

```
#include <sys/condvar.h>

void
cv_init(kcondvar_t *cv, const char *wmesg);

void
cv_destroy(kcondvar_t *cv);

void
cv_wait(kcondvar_t *cv, kmutex_t *mtx);

int
cv_wait_sig(kcondvar_t *cv, kmutex_t *mtx);

int
cv_timedwait(kcondvar_t *cv, kmutex_t *mtx, int ticks);

int
cv_timedwait_sig(kcondvar_t *cv, kmutex_t *mtx, int ticks);

void
cv_signal(kcondvar_t *cv);

void
cv_broadcast(kcondvar_t *cv);

bool
cv_has_waiters(kcondvar_t *cv);

options DIAGNOSTIC
options LOCKDEBUG
```

DESCRIPTION

Condition variables (CVs) are used in the kernel to synchronize access to resources that are limited (for example, memory) and to wait for pending I/O operations to complete.

The *kcondvar_t* type provides storage for the CV object. This should be treated as an opaque object and not examined directly by consumers.

OPTIONS

options DIAGNOSTIC

Kernels compiled with the DIAGNOSTIC option perform basic sanity checks on CV operations.

options LOCKDEBUG

Kernels compiled with the LOCKDEBUG option perform potentially CPU intensive sanity checks on CV operations.

FUNCTIONS

cv_init(*cv*, *wmesg*)

Initialize a CV for use. No other operations can be performed on the CV until it has been initialized.

The *wmesg* argument specifies a string of no more than 8 characters that describes the resource or condition associated with the CV. The kernel does not use this argument directly but makes it available for utilities such as ps(1) to display.

cv_destroy(*cv*)

Release resources used by a CV. The CV must not be in use when it is destroyed, and must not be used afterwards.

cv_wait(*cv*, *mtx*)

Cause the current LWP to wait non-interruptably for access to a resource, or for an I/O operation to complete. The LWP will resume execution when awoken by another thread using **cv_signal**() or **cv_broadcast**().

mtx specifies a kernel mutex to be used as an interlock, and must be held by the calling LWP on entry to **cv_wait**(). It will be released once the LWP has prepared to sleep, and will be reacquired before **cv_wait**() returns.

A small window exists between testing for availability of a resource and waiting for the resource with **cv_wait**(), in which the resource may become available again. The interlock is used to guarentee that the resource will not be signalled as available until the calling LWP has begun to wait for it.

Non-interruptable waits have the potential to deadlock the system, and so must be kept short (typically, under one second).

cv_wait_sig(*cv*, *mtx*)

As per **cv_wait**(), but causes the current LWP to wait interruptably. If the LWP recieves a signal, or is interrupted by another condition such as its containing process exiting, the wait is ended early and an error code returned.

If **cv_wait_sig**() returns as a result of a signal, the return value is ERESTART if the signal has the SA_RESTART property. If awoken normally, the value is zero, and EINTR under all other conditions.

cv_timedwait(*cv*, *mtx*, *ticks*)

As per **cv_wait**(), but will return early if a timeout specified by the *ticks* argument expires.

ticks is an architecture and system dependent value related to the number of clock interrupts per second. See hz(9) for details. The mstohz(9) macro can be used to convert a timeout expressed in milliseconds to one suitable for **cv_timedwait**(). If the *ticks* argument is zero, **cv_timedwait**() behaves exactly like **cv_wait**().

If the timeout expires before the LWP is awoken, the return value is EWOULDBLOCK. If awoken normally, the return value is zero.

cv_timedwait_sig(*cv*, *mtx*, *ticks*)

As per **cv_wait_sig**(), but also accepts a timeout value and will return EWOULDBLOCK if the timeout expires.

cv_signal(*cv*)

Awaken one LWP (potentially among many) that is waiting on the specified condition variable. The mutex passed to the wait function (*mtx*) must also be held when calling **cv_signal**().

(Note that **cv_signal**() is erroneously named in that it does not send a signal in the traditional sense to LWPs waiting on a CV.)

cv_broadcast(*cv*)

Awaken all LWPs waiting on the specified condition variable. The mutex passed to the wait function (*mtx*) must also be held when calling **cv_broadcast**().

cv_has_waiters(*cv*)

Return `true` if one or more LWPs are waiting on the specified condition variable.

cv_has_waiters() cannot test reliably for interruptable waits. It should only be used to test for non-interruptable waits made using **cv_wait**().

cv_has_waiters() should only be used when making diagnostic assertions, and must be called while holding the interlocking mutex passed to **cv_wait**().

EXAMPLES

Consuming a resource:

```
/*
 * Lock the resource.  Its mutex will also serve as the
 * interlock.
 */
mutex_enter(&res->mutex);

/*
 * Wait for the resource to become available.
 */
while (res->state == BUSY)
        cv_wait(&res->condvar, &res->mutex);

/*
 * It's now available to us.  Take ownership of the
 * resource, and consume it.
 */
res->state = BUSY;
mutex_exit(&res->mutex);
consume(res);
```

Releasing a resource for the next consumer to use:

```
mutex_enter(&res->mutex);
res->state = IDLE;
cv_signal(&res->condvar);
mutex_exit(&res->mutex);
```

CODE REFERENCES

This section describes places within the NetBSD source tree where code implementing condition variables can be found. All pathnames are relative to `/usr/src`.

The core of the CV implementation is in `sys/kern/kern_condvar.c`.

The header file `sys/sys/condvar.h` describes the public interface.

SEE ALSO

`sigaction`(2), `errno`(9), `mb`(9), `mstohz`(9), `mutex`(9), `rwlock`(9)

Jim Mauro and Richard McDougall, *Solaris Internals: Core Kernel Architecture, Prentice Hall*, 2001, ISBN 0-13-022496-0.

HISTORY

The CV primitives first appeared in NetBSD 5.0.

NAME

config — the autoconfiguration framework "device definition" language

DESCRIPTION

In NetBSD, the config(1) program reads and verifies a machine description file (documented in config(5)) that specifies the devices to include in the kernel. A table is produced by config(1) which is used by the kernel during autoconfiguration (see autoconf(9)) giving needed hints and details on matching hardware devices with device drivers.

Each device in the machine description file has:

A Name The name is simply an alphanumeric string that ends in a unit number (e.g., "sd0", "sd1", and so forth). These unit numbers identify particular instances of a base device name; the base name in turn maps directly to a device driver. Device unit numbers are independent of hardware features.

A Parent Every device must have a parent. The pairing is denoted by "child at parent". These pairings form the links in a directed graph. The root device is the only exception, as it does not have a parent.

Locators Locators are used to augment the parent/child pairings that locate specific devices. Each locator value is simply an integer that represents some sort of device address on the parent bus or controller. This can be a memory address, an I/O port, a driver number, or any other value. Locators can sometimes be wildcarded on devices that support direct connection.

Attributes

An attribute describes the device's relationship with the code; it then serves to constrain the device graph. A *plain attribute* describes some attribute of a device. An *interface attribute* describes a kind of "software interface" and declares the device's ability to support other devices that use that interface. In addition, an interface attribute usually identifies additional locators.

During autoconfiguration, the directed graph is turned into a tree by nominating one device as the root node and matching drivers with devices by doing a depth-first traversal through the graph starting at this root node.

However, there must be constraints on the parent/child pairings that are possible. These constraints are embedded in the "device definition" files. The remainder of this page describes the "device definition" language and how to use this language to add new functionality to the NetBSD kernel.

DEVICE DEFINITION FILES

The device definition files are separated into machine-dependent and machine-independent files. The machine-dependent file is located in sys/arch/<arch>/conf/files.<arch>, where <arch> is the name of NetBSD architecture. The machine-independent file is located in sys/conf/files. It in turn includes files for the machine-independent drivers located in sys/dev/<bus>/files.<bus>, where <bus> is the name of the machine-independent bus.

These files define all legal devices and pseudo-devices. They also define all attributes and interfaces, establishing the rules that determine allowable machine descriptions, and list the source files that make up the kernel.

Each device definition file consists of a list of statements, typically one per line. Comments may be inserted anywhere using the "#" character, and any line that begins with white space continues the previous line. Valid statements are:

cinclude filename

Conditionally include contents of file *filename* to currently processed configuration. If the specified *filename* doesn't exist, a warning is printed, but the error ignored.

defflag [filename] {options}
: The space-separated list of pre-processor macros *options* are defined in file *filename*. This statement permits ''options FOO'' for FOO (i.e, without a value) in the machine description file. `config`(1) will generate an error if a value is given. If the filename field is not specified, it will be constructed based upon the lower-case of the option name, ''opt_foo.h'' for example. The *option* is case-sensitive.

defparam [filename] {options}
: The space-separated list of pre-processor macros *options* are defined in file *filename*. This statement permits ''options FOO=bar'' or ''option FOO="\"com\""'' in the machine description file. `config`(1) will generate an error if a value is not given. If the filename field is not specified, it will be constructed based upon the lower-case of the option name, ''opt_foo.h'' for example. The *option* is case-sensitive.

defopt [filename] {options}
: The space-separated list of pre-processor macros *options* are defined in file *filename*. This statement permits the syntax of either the defflag and defparam statements and `config`(1) does not perform any checking of whether ''options FOO'' takes a value. Therefore, the use of the defopt statement is deprecated in favour of the defflag and defparam statements. If the filename field is not specified, it will be constructed based upon the lower-case of the option name, ''opt_foo.h'' for example. The *option* is case-sensitive.

deffs [filename] name
: Define a filesystem *name*.

devclass name
: Define a device class *name*. A device class is similar to an attribute.

define name [{locators}]
: The attribute *name* is defined and device definitions can then refer to it. If the attribute is an interface attribute and defines optional *locators*, device attributes that refer to *name* are assumed to share the interface and require the same locators.

device name [{locators}]: [attributes]
: The device *name* is defined and requires the optional comma-separated list of locators *locators*. The optional *attributes* define attribute dependencies.

attach name at interface [with ifname]: [attributes]
: The device *name* is defined and supports the interface *interface*. If *ifname* is specified, it is used to specify the interface to the driver for device *name* (see `driver`(9) for details). The optional *attributes* define attribute dependencies.

defpseudo name: [{locators}]
: The pseudo-device *name* is defined. The optional *locators* may be defined, but are largely pointless since no device can attach to a pseudo-device.

file pathname [attributes [flags]] [rule]
: The file *pathname* is added to the list of files used to build the kernel. If no attributes are specified, the file is always added to the kernel compilation. If any of the attributes are specified by other devices in the machine description file, then the file is included in compilation, otherwise it is omitted. Valid values for the optional flags are:

  needs-count
  : Specify that config should generate a file for each of the attributes notifying the driver how many of some particular device or set of devices are configured in the kernel. This flag allows drivers to make calculations of driver used at compile time. This option prevents autoconfiguration cloning.

needs-flag
: This flag performs the same operation as *needs-count* but only records if the number is nonzero. Since the count is not exact, *needs-flag* does not prevent autoconfiguration cloning.

device-major name char [block] [attributes]
: The character device switch *name* associated with a character major device number is added to the list of device switches used to build the kernel. If *block* is specified, the block device switch associated with a block major device number is also added. If all of attributes are specified by devices in the machine description files, then device switches are added into the device switches' table of the kernel in compilation, otherwise they are omitted.

include *filename*
: Include contents of file *filename* to currently processed configuration. If the specified *filename* doesn't exist, `config`(1) exits with error.

package *filename*
: Changes prefix to directory of *filename*, processes contents of *filename*, and switches back to previous prefix. This is syntactic sugar for:
  prefix *dirname(filename)*
  include *basename(filename)*
  prefix

prefix [*dirname*]
: If *dirname* is specified, it is pushed on top of prefix stack. Any further files specified via option *file* would have the prefix implicitly prepended before its *filename*. If *dirname* is not specified, pops (removes) a prefix from prefix stack.

To allow locators to be wildcarded in the machine description file, their default value must be defined in the attribute definition. To allow locators to be omitted entirely in the machine description file, enclose the locator in square brackets. This can be used when some locators do not make sense for some devices, but the software interface requires them.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the autoconfiguration framework can be found. All pathnames are relative to `/usr/src`.

The device definition files are in `sys/conf/files`, `sys/arch/<arch>/conf/files.<arch>`, and `sys/dev/<bus>/files.<bus>`.

The grammar for machine description files can be found in `config`(1), in `usr.bin/config/gram.y`.

SEE ALSO

`config`(1), `config`(5), `autoconf`(9), `driver`(9)

Building 4.4 BSD Systems with Config.

Chris Torek, *Device Configuration in 4.4BSD*, 1992.

HISTORY

Autoconfiguration first appeared in 4.1BSD. The autoconfiguration framework was completely revised in 4.4BSD. It has been modified within NetBSD to support bus-independent drivers and bus-dependent attachments.

NAME

cnbell, **cnflush**, **cngetc**, **cngetsn**, **cnhalt**, **cnpollc**, **cnputc** — console access interface

SYNOPSIS

```
#include <dev/cons.h>

void
cnbell(u_int pitch, u_int period, u_int volume);

void
cnflush(void);

int
cngetc(void);

int
cngetsn(char *cp, int size);

void
cnhalt(void);

void
cnpollc(int on);

void
cnputc(int c);
```

DESCRIPTION

These functions operate over the current console device. The console must be initialized before these functions can be used.

Console input polling functions **cngetc**(), **cngetsn**() and **cnpollc**() are only to be used during initial system boot, e.g., when asking for root and dump device or to get necessary user input within mountroothooks. Once the system boots, user input is read via standard tty(4) facilities.

The following is a brief description of each function:

cnbell() Ring a bell at appropriate *pitch*, for duration of *period* milliseconds at given *volume*. Note that the *volume* value is ignored commonly.

cnflush() Waits for all pending output to finish.

cngetc() Poll (busy wait) for an input and return the input key. Returns 0 if there is no console input device. **cnpollc**() *must* be called before **cngetc**() could be used. **cngetc**() should be used during kernel startup only.

cngetsn() Read one line of user input, stop reading once the newline key is input. Input is echoed back. This uses **cnpollc**() and **cngetc**(). Number of read characters is *size* at maximum, user is notified by console bell when the end of input buffer is reached. <Backspace> key works as expected. <@> or <CTRL>-u make **cngetsn**() discard input read so far, print newline and wait for next input. **cngetsn**() returns number of characters actually read, excluding the final newline. *cp* is *not* zero-ended before return. **cngetsn**() should be used during kernel startup only.

cnhalt() Terminates the console device (i.e. cleanly shuts down the console hardware.)

cnpollc() Switch the console driver to polling mode if *on* is nonzero, or back to interrupt driven mode if *on* is zero. **cnpollc**() should be used during kernel startup only.

cnputc() Console kernel output character routine. Commonly, kernel code uses printf(9) rather than using this low-level interface.

EXAMPLES

This waits until a <Enter> key is pressed:

```
int c;

cnpollc(1);
for(;;) {
        c = cngetc();
        if ((c == '\r' || (c == '\n')) {
                printf("\n");
                break;
        }
}
cnpollc(0);
```

SEE ALSO

pckbd(4), pcppi(4), tty(4), wscons(4), wskbd(4), printf(9), spl(9), wscons(9)

NAME

copy, **copyin**, **copyout**, **copystr**, **copyinstr**, **copyoutstr** — kernel space to/from user space copy functions

SYNOPSIS

```
#include <sys/types.h>
#include <sys/systm.h>

int
copyin(const void *uaddr, void *kaddr, size_t len);

int
copyout(const void *kaddr, void *uaddr, size_t len);

int
copystr(const void *kfaddr, void *kdaddr, size_t len, size_t *done);

int
copyinstr(const void *uaddr, void *kaddr, size_t len, size_t *done);

int
copyoutstr(const void *kaddr, void *uaddr, size_t len, size_t *done);

int
copyin_proc(struct lwp *l, const void *uaddr, void *kaddr, size_t len);

int
copyout_proc(struct lwp *l, const void *kaddr, void *uaddr, size_t len);

int
ioctl_copyin(int ioctlflags, const void *src, void *dst, size_t len);

int
ioctl_copyout(int ioctlflags, const void *src, void *dst, size_t len);
```

DESCRIPTION

The **copy** functions are designed to copy contiguous data from one address to another. All but **copystr**() copy data from user-space to kernel-space or vice-versa.

The **copy** routines provide the following functionality:

copyin() Copies *len* bytes of data from the user-space address *uaddr* in the current process to the kernel-space address *kaddr*.

copyout() Copies *len* bytes of data from the kernel-space address *kaddr* to the user-space address *uaddr* in the current process.

copystr() Copies a NUL-terminated string, at most *len* bytes long, from kernel-space address *kfaddr* to kernel-space address *kdaddr*. If the *done* argument is non-NULL, the number of bytes actually copied, including the terminating NUL, is returned in **done*.

copyinstr() Copies a NUL-terminated string, at most *len* bytes long, from user-space address *uaddr* in the current process to kernel-space address *kaddr*. If the *done* argument is non-NULL, the number of bytes actually copied, including the terminating NUL, is returned in **done*.

copyoutstr() Copies a NUL-terminated string, at most *len* bytes long, from kernel-space address *kaddr* to user-space address *uaddr* in the current process. If the *done* argument is non-NULL, the number of bytes actually copied, including the terminating NUL, is returned in ∗*done*.

copyin_proc() Like **copyin**(), except it operates on the address space of the lwp *l*.

copyout_proc() Like **copyout**(), except it operates on the address space of the lwp *l*.

ioctl_copyin() Like **copyin**(), except it operates on kernel adresses when the FKIOCTL flag is passed in *ioctlflags* from the ioctl call.

ioctl_copyout()
Like **copyout**(), except it operates on kernel adresses when the FKIOCTL flag is passed in *ioctlflags* from the ioctl call.

RETURN VALUES

The **copy** functions return 0 on success or EFAULT if a bad address is encountered. In addition, the **copystr**(), **copyinstr**(), and **copyoutstr**() functions return ENAMETOOLONG if the string is longer than *len* bytes.

SEE ALSO

fetch(9), store(9)

NAME

coredump_write — common coredump write routine

SYNOPSIS

```
#include <sys/signalvar.h>

int
coredump_write(void *iocookie, enum uio_seg segflg, const void *data,
    size_t len);
```

DESCRIPTION

coredump_write() is used by both machine-dependent and machine-independent components to write information to a coredump. *iocookie* is an opaque pointer that was supplied to the caller of **coredump_write**(). *segflg* indicates where the *data* is located, system space or user space. *data* points to the information to be written to the coredump. *len* is the amount of data to be written.

coredump_write() returns 0 on success and an appropriate error code on failure.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using **coredump_write**() can be found. All pathnames are relative to `/usr/src`.

Process core dumps are initiated within the file `sys/kern/kern_sig.c`. Process core dumps for ELF NetBSD binaries are performed within the files `sys/kern/core_elf32.c` or `sys/kern/core_elf64.c`. Process core dumps for other NetBSD binaries are performed within the file `sys/kern/core_netbsd.c`.

SEE ALSO

`cpu_coredump`(9)

NAME

cpu_configure — machine-dependent device autoconfiguration

SYNOPSIS

```
#include <sys/systm.h>

void
cpu_configure(void);
```

DESCRIPTION

cpu_configure() is called during system bootstrap to perform the machine-dependent portion of device autoconfiguration. It sets the configuration machinery in motion by finding the root bus ("mainbus"). When this function returns, interrupts must be enabled.

cpu_configure() performs the following tasks:

- initialize soft interrupts (see softintr(9))
- initialize CPU interrupts and SPLs
- call **config_rootfound**() for "mainbus"
- complete any initialization deferred from **cpu_startup**().

SEE ALSO

autoconf(9), cpu_startup(9)

NAME

cpu_coredump — machine-dependent process core dump interface

SYNOPSIS

```
#include <sys/signalvar.h>

int
cpu_coredump(struct lwp *l, void *iocookie, struct core *chdr);

int
cpu_coredump32(struct lwp *l, void *iocookie, struct core32 *chdr);
```

DESCRIPTION

cpu_coredump() is the machine-dependent interface invoked by machine-independent code to dump the machine-dependent header information at the start of a process core dump. The header information primarily consists of the CPU and floating-point registers. *l* is the lwp structure of the thread being dumped. *iocookie* is an opaque pointer to be passed to **coredump_write**(). Information about the machine-dependent header sections are returned in *chdr*.

cpu_coredump() returns 0 on success and an appropriate error code on failure.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-dependent coredump interface can be found. All pathnames are relative to `/usr/src`.

Process core dumps are initiated within the file `sys/kern/kern_sig.c`. Process core dumps for ELF NetBSD binaries are performed within the files `sys/kern/core_elf32.c` or `sys/kern/core_elf64.c`. Process core dumps for other NetBSD binaries are performed within the file `sys/kern/core_netbsd.c`.

SEE ALSO

`coredump_write`(9)

NAME

cpu_dumpconf, **cpu_dump**, **cpu_dumpsize**, **dumpsys** — machine-dependent kernel core dumps

SYNOPSIS

```
#include <sys/types.h>
#include <sys/systm.h>

void
cpu_dumpconf(void);

int
cpu_dump(int (*dump)(dev_t, daddr_t, void *, size_t), daddr_t *blknop);

int
cpu_dumpsize(void);

void
dumpsys(void);
```

DESCRIPTION

cpu_dumpconf() is the machine-dependent interface invoked during system bootstrap to determine the dump device and initialize machine-dependent kernel core dump state. Internally, **cpu_dumpconf**() will invoke **cpu_dumpsize**() to calculate the size of machine-dependent kernel core dump headers.

dumpsys() is invoked by **cpu_reboot**() to dump kernel physical memory onto the dump device. **dumpsys**() invokes **cpu_dump**() to write the machine-dependent header to the dump device at block number **blknop* using the dump device's PIO dump routine specified by the *dump* argument.

cpu_dumpsize(), **cpu_dump**(), and **dumpsys**() are parts of the machine-dependent interface, however they are not exported to machine-independent code.

SEE ALSO

cpu_reboot(9)

NAME

cpu_idle — machine-dependent processor idling interface

SYNOPSIS

```
#include <sys/cpu.h>

void
cpu_idle(void);
```

DESCRIPTION

cpu_idle() is called by machine-independent code when the processor has nothing to do. It can be used to conserve the processor power, for example.

cpu_idle() returns immediately if **cpu_need_resched**() has been called for the processor after the last call of **cpu_idle**() or **cpu_did_resched**() on the processor. **cpu_idle**() returns as soon as possible when **cpu_need_resched**() is called for the processor. Otherwise, it returns whenever it likes.

cpu_idle() is called at IPL_NONE, without any locks held.

EXAMPLES

The simplest (and, in some cases, the best) implementation of **cpu_idle**() is the following.

```
void
cpu_idle(void)
{
        /* nothing */
}
```

SEE ALSO

cpu_need_resched(9), cpu_switchto(9), intro(9), spl(9)

NAME

cpu_initclocks — machine-dependent clock setup interface

SYNOPSIS

```
#include <sys/systm.h>

void
cpu_initclocks(void);
```

DESCRIPTION

cpu_initclocks() is invoked by **initclocks**() during system bootstrap, immediately after autoconfiguration, to perform the machine-dependent initialization of clock frequencies and start the real-time and statistic clocks running.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-dependent clocks initialization interface can be found. All pathnames are relative to `/usr/src`.

Machine-independent clock interface operations are performed within the file `sys/kern/kern_clock.c`.

SEE ALSO

`autoconf`(9)

NAME

cpu_lwp_fork, **child_return**, **proc_trampoline** — finish a fork operation

SYNOPSIS

```
#include <sys/proc.h>

void
cpu_lwp_fork(struct lwp *l1, struct lwp *l2, void *stack, size_t stacksize,
    void (*func)(void *), void *arg);

void
child_return(void *arg);
```

DESCRIPTION

cpu_lwp_fork() is the machine-dependent portion of **fork1**() which finishes a fork operation, with child lwp *l2* nearly set up. It copies and updates the PCB and trap frame from the parent *l1*, making the child ready to run.

cpu_lwp_fork() rigs the child's kernel stack so that it will start in **proc_trampoline**(). **proc_trampoline**() does not have a normal calling sequence and is entered by **cpu_switch**(). If an alternate user-level stack is requested (with non-zero values in both the *stack* and *stacksize* arguments), the user stack pointer is set up accordingly.

After being entered by **cpu_switch**() and while running in user context (within the kernel) **proc_trampoline**() will invoke the function *func* with the argument *arg*. If a kernel thread is being created, the return path and argument are specified with *func* and *arg*. If a user process is being created, **fork1**() will pass **child_return**() and *l2* to **cpu_lwp_fork**() as *func* and *arg* respectively. This causes the newly-created child process to go directly to user level with an apparent return value of 0 from fork(2), while the parent process returns normally.

SEE ALSO

fork(2), cpu_switch(9), fork1(9)

NAME

cpu_need_resched — context switch notification

SYNOPSIS

```
#include <sys/cpu.h>

void
cpu_need_resched(struct cpu_info *ci, int flags);
```

DESCRIPTION

The **cpu_need_resched**() function is the machine-independent interface for the scheduler to notify machine-dependent code that a context switch from the current LWP, on the cpu *ci*, is required. This event may occur if a higher priority LWP appears on the run queue or if the current LWP has exceeded its time slice.

If RESCHED_IMMED flag is specified in *flags*, machine-dependent code should make a context switch happen as soon as possible. In that case, for example, if *ci* is not the current processor, **cpu_need_resched**() typically issues an inter processor call to the processor to make it notice the need of a context switch as soon as possible.

EXAMPLES

Specifically, the **cpu_need_resched**() function will perform the following operations:

- Set a per-processor flag which is checked by userret(9) when returning to user-mode execution.
- Post an asynchronous software trap (AST).
- Send an inter processor interrupt to wake up **cpu_idle**().

SEE ALSO

sched_4bsd(9), userret(9)

NAME

cpu_number — unique CPU identification number

SYNOPSIS

```
#include <sys/types.h>
#include <machine/cpu.h>

cpuid_t
cpu_number(void);
```

DESCRIPTION

cpu_number() returns the unique CPU identification number for the CPU that this thread is running on.

SEE ALSO

curcpu(9)

NAME

cpu_reboot — halt or reboot the system

SYNOPSIS

```
#include <sys/reboot.h>

void
cpu_reboot(int howto, char *bootstr);
```

DESCRIPTION

The **cpu_reboot**() function handles final system shutdown, and either halts or reboots the system. The exact action to be taken is determined by the flags passed in *howto* and by whether or not the system has finished autoconfiguration.

If the system has finished autoconfiguration, **cpu_reboot**() does the following:

1. Sets the *boothowto* system variable from the *howto* argument.
2. If this is the first invocation of **cpu_reboot**() and the RB_NOSYNC flag is not set in *howto*, syncs and unmounts the system disks by calling vfs_shutdown(9) and sets the time of day clock by calling resettodr(9).
3. Disables interrupts.
4. If rebooting after a crash (i.e., if RB_DUMP is set in *howto*, but RB_HALT is not), saves a system crash dump.
5. Runs any shutdown hooks by calling doshutdownhooks(9).
6. Prints a message indicating that the system is about to be halted or rebooted.
7. If RB_HALT is set in *howto*, halts the system. Otherwise, reboots the system.

If the system has not finished autoconfiguration, **cpu_reboot**() runs any shutdown hooks by calling doshutdownhooks(9), prints a message, and halts the system.

If RB_STRING is set in howto, then the parameter *bootstr* is passed to the system boot loader on some ports.

SEE ALSO

doshutdownhooks(9), dumpsys(9), resettodr(9), vfs_shutdown(9)

NAME

cpu_rootconf — machine-dependent root file system setup

SYNOPSIS

```
#include <sys/types.h>
#include <sys/systm.h>

void
cpu_rootconf(void);
```

DESCRIPTION

cpu_rootconf() is the machine-dependent interface invoked during system bootstrap to determine the root file system device and initialize machine-dependent file system state. **cpu_rootconf**() invokes the machine-independent function **setroot**() to record the boot/root device and the boot partition information for use in machine-independent code.

SEE ALSO

setroot(9)

NAME

cpu_startup — machine-dependent CPU startup

SYNOPSIS

```
#include <sys/systm.h>

void
cpu_startup(void);
```

DESCRIPTION

cpu_startup() is invoked early during system bootstrap, after the console has been set up and immediately after uvm(9) has been initialized. **cpu_startup**() performs the following tasks:

- prints the initial copyright message
- allocate memory and buffers for kernel tables
- initialize the CPU

SEE ALSO

autoconf(9), uvm(9)

NAME

cpu_swapout, **cpu_swapin** — machine-dependent swap interface

SYNOPSIS

```
#include <sys/lwp.h>
#include <machine/cpu.h>

void
cpu_swapout(struct lwp *l);

void
cpu_swapin(struct lwp *l);
```

DESCRIPTION

cpu_swapout() and **cpu_swapin**() are the machine-dependent interface for swapping processes in and out of the system. They perform any machine-specific operations such as saving and restoring floating point state. They are invoked by **uvm_swapout**() and **uvm_swapin**() respectively.

SEE ALSO

uvm(9)

NAME

cpu_switchto — machine-dependent LWP context switching interface

SYNOPSIS

```
#include <sys/cpu.h>

lwp_t *
cpu_switchto(lwp_t *oldlwp, lwp_t *newlwp, bool returning);
```

DESCRIPTION

cpu_switchto() saves the context of the LWP which is currently running on the processor, and restores the context of the LWP specified by *newlwp*.

cpu_switchto() doesn't switch address spaces.

cpu_switchto() sets curlwp to *newlwp*.

cpu_switchto() should be called at IPL_SCHED. When **cpu_switchto**() returns, the caller should lower the priority level as soon as possible.

cpu_switchto() might be called with spin mutexes held.

It takes the following arguments.

oldlwp Specify the lwp from which we are going to switch, i.e., the calling LWP. If it was NULL, the context of the LWP currently running on this processor is not saved.

newlwp Specify the lwp to which we are going to switch. It must not be NULL.

returning
Only meaningful if the architecture implements fast software interrupts. If true, it indicates that *oldlwp* is a soft interrupt LWP that is blocking. It's a good indication that any kind of address space or user activity can be completely ignored. For example: **ras_lookup**(), cache flushes, TLB wirings, adjusting lazy FPU state. All that is required is to restore the register state and stack, and return to the interrupted LWP.

RETURN VALUES

cpu_switchto() does not return until another LWP calls **cpu_switchto**() to switch to us. It returns the oldlwp argument of the **cpu_switchto**() which is called to switch back to our LWP. It's either an LWP which called *cpu_switchto* to switch to us or NULL in the case that the LWP was exiting.

SEE ALSO

swapcontext(3), intro(9), mutex(9), spl(9)

NAME

CSF — The NetBSD common scheduler framework

SYNOPSIS

```
#include <sys/sched.h>

void
sched_rqinit(void);

void
sched_setup(void);

void
sched_cpuattach(struct cpu_info *);

void
sched_tick(struct cpu_info *);

void
sched_schedclock(lwp_t *);

bool
sched_curcpu_runnable_p(void);

lwp_t *
sched_nextlwp(void);

void
sched_enqueue(lwp_t *, bool);

void
sched_dequeue(lwp_t *);

void
sched_nice(struct proc *, int);

void
sched_proc_fork(struct proc *, struct proc *);

void
sched_proc_exit(struct proc *, struct proc *);

void
sched_lwp_fork(lwp_t *);

void
sched_lwp_exit(lwp_t *);

void
sched_setrunnable(lwp_t *);

void
sched_print_runqueue(void (*pr)(const char *, ...));

void
sched_pstats_hook(struct proc *, int);

void
sched_pstats(void *arg);
```

pri_t
sched_kpri(*lwp_t **);

void
resched_cpu(*lwp_t **);

void
setrunnable();

void
schedclock(*lwp_t **);

void
sched_init(*void*);

DESCRIPTION

CSF provides a modular and self-contained interface for implementing different thread scheduling algorithms. The different schedulers can be selected at compile-time. Currently, the only scheduler available is sched_4bsd(9), the traditional 4.4BSD thread scheduler.

The interface is divided into two parts: A set of functions each scheduler needs to implement and common functions used by all schedulers.

Scheduler-specific functions

The following functions have to be implemented by the individual scheduler.

Scheduler initialization

void **sched_cpuattach**(*struct cpu_info **)
Per-CPU scheduler initialization routine.

void **sched_rqinit**(*void*)
Initialize the scheduler's runqueue data structures.

void **sched_setup**(*void*)
Setup initial scheduling parameters and kick off timeout driven events.

Runqueue handling

Runqueue handling is completely internal to the scheduler. Other parts of the kernel should access runqueues only through the following functions:

void **sched_enqueue**(*lwp_t **, *bool*)
Place an LWP within the scheduler's runqueue structures.

void **sched_dequeue**(*lwp_t **)
Remove an LWP from the scheduler's runqueue structures.

*lwp_t ** **sched_nextlwp**(*void*)
Return the LWP that should run the CPU next.

bool **sched_curcpu_runnable_p**(*void*)
Indicate if there is a runnable LWP for the current CPU.

void **sched_print_runqueue**(*void (*pr)(const char *, ...)*)
Print runqueues in DDB.

Core scheduler functions

void **sched_tick**(*struct cpu_info* *)
Periodically called from `hardclock`(9). Determines if a reschedule is necessary, if the running LWP has used up its quantum.

void **sched_schedclock**(*lwp_t* *)
Periodically called from **schedclock**() in order to handle priority adjustment.

Priority adjustment

void **sched_nice**(*struct proc* *, *int*)
Recalculate the process priority according to its nice value.

General helper functions

void **sched_proc_fork**(*struct proc* *, *struct proc* *)
Inherit the scheduling history of the parent process after **fork**().

void **sched_proc_exit**(*struct proc* *, *struct proc* *)
Charge back a processes parent for its resource usage.

void **sched_lwp_fork**(*lwp_t* *)
LWP-specific version of the above

void **sched_lwp_exit**(*lwp_t* *)
LWP-specific version of the above

void **sched_setrunnable**(*lwp_t* *)
Scheduler-specific actions for **setrunnable**().

void **sched_pstats_hook**(*struct proc* *, *int*)
Scheduler-specific actions for **sched_pstats**().

Common scheduler functions

pri_t **sched_kpri**(*lwp_t* *)
Scale a priority level to a kernel priority level, usually for an LWP that is about to sleep.

void **sched_pstats**(*void* *)
Update process statistics and check CPU resource allocation.

inline void **resched_cpu**(*lwp_t* *)
Arrange for a reschedule.

void **setrunnable**(*lwp_t* *)
Change process state to be runnable, placing it on a runqueue if it is in memory, awakening the swapper otherwise.

void **schedclock**(*lwp_t* *)
Scheduler clock. Periodically called from **statclock**().

void **sched_init**(*void*)
Initialize callout for **sched_pstats**() and call **sched_setup**() to initialize any other scheduler-specific data.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing the scheduler can be found. All pathnames are relative to `/usr/src`.

The **CSF** programming interface is defined within the file `sys/sys/sched.h`.

Functions common to all scheduler implementations are in `sys/kern/kern_synch.c`.

The traditional 4.4BSD scheduler is implemented in `sys/kern/sched_4bsd.c`.

SEE ALSO

`mi_switch`(9), `preempt`(9), `sched_4bsd`(9)

HISTORY

The **CSF** appeared in NetBSD 5.0.

AUTHORS

The **CSF** was written by Daniel Sieger ⟨dsieger@NetBSD.org⟩.

NAME

curproc, **curcpu** — current process and processor

SYNOPSIS

```
#include <sys/proc.h>

struct cpu_info *
curcpu(void);

struct proc *
curproc(void);
```

DESCRIPTION

curcpu() returns a pointer to a *cpu_info* structure containing information of the CPU that this thread is running on. **curproc**() returns a pointer to the process currently running on this CPU.

NAME

ddc — VESA Display Data Channel V2

SYNOPSIS

```
#include <dev/i2c/ddcvar.h>

int
ddc_read_edid(i2c_tag_t tag, uint8_t *dest, size_t len);
```

DESCRIPTION

The **ddc_read_edid**() reads a VESA Extended Display Identification Data block (EDID) via VESA Display Data Channel (DDCv2). DDCv2 is a protocol for data exchange between display devices (such as monitors and flat panels) and host machines using an I2C bus.

The *tag* argument is a machine-dependent tag used to specify the I2C bus on which the DDCv2 device is located. The *dest* argument is a pointer to a buffer where the EDID data will be stored. The *len* argument is the amount of data to read into the buffer. (The buffer must be large enough.) Typically, this value will be 128, which is the size of a normal EDID data block.

Normally the EDID data block will be post-processed with the **edid_parse**() function.

RETURN VALUES

The **ddc_read_edid**() function returns zero on success, and non-zero otherwise.

ENVIRONMENT

The **ddc_read_edid**() function is part of the ddc(4) driver, and is only included in the kernel if that driver is also included.

EXAMPLES

The following code uses **ddc_read_edid**() to retrieve and print information about a monitor:

```
struct edid_info info;
i2c_tag_t        tag;
char            buffer[128];

...
/* initialize i2c tag... */
...
if ((ddc_read_edid(tag, buffer, 128) == 0) &&
    (edid_parse(buffer, &info) == 0))
        edid_print(info);
...
```

Note that this must be called before the PCI bus is attached during autoconfiguration.

SEE ALSO

ddc(4), edid(9), iic(9)

HISTORY

DDCv2 support was added in NetBSD 4.0.

AUTHORS

Garrett D'Amore ⟨gdamore@NetBSD.org⟩

NAME

delay, **DELAY** — microsecond delay

SYNOPSIS

```
#include <machine/param.h>

void
delay(unsigned int us);

void
DELAY(unsigned int us);
```

DESCRIPTION

Wait approximately *us* microseconds.

The delay is implemented as a machine loop, preventing events other than interrupt handlers for unmasked interrupts to run. **DELAY**() is reentrant (doesn't modify any global kernel or machine state) and is safe to use in interrupt or process context.

For long delays, condition variables should be considered, however they can only be used from process context and their resolution is limited by the system clock frequency.

SEE ALSO

condvar(9), hz(9)

NAME

disk, **disk_init**, **disk_attach**, **disk_detach**, **disk_destroy**, **disk_busy**, **disk_unbusy**, **disk_find**, **disk_blocksize** — generic disk framework

SYNOPSIS

```
#include <sys/types.h>
#include <sys/disklabel.h>
#include <sys/disk.h>

void
disk_init(struct disk *, const char *name, const struct dkdriver *driver);

void
disk_attach(struct disk *);

void
disk_detach(struct disk *);

void
disk_destroy(struct disk *);

void
disk_busy(struct disk *);

void
disk_unbusy(struct disk *, long bcount, int read);

struct disk *
disk_find(const char *);

void
disk_blocksize(struct disk *, int blocksize);
```

DESCRIPTION

The NetBSD generic disk framework is designed to provide flexible, scalable, and consistent handling of disk state and metrics information. The fundamental component of this framework is the **disk** structure, which is defined as follows:

```
struct disk {
        TAILQ_ENTRY(disk) dk_link;    /* link in global disklist */
        const char     *dk_name;      /* disk name */
        prop_dictionary_t dk_info;    /* reference to disk-info dictionary */
        int            dk_bopenmask;  /* block devices open */
        int            dk_copenmask;  /* character devices open */
        int            dk_openmask;   /* composite (bopen|copen) */
        int            dk_state;      /* label state   ### */
        int            dk_blkshift;   /* shift to convert DEV_BSIZE to blks */
        int            dk_byteshift;  /* shift to convert bytes to blks */

        /*
         * Metrics data; note that some metrics may have no meaning
         * on certain types of disks.
         */
        struct io_stats       *dk_stats;

        const struct dkdriver *dk_driver;     /* pointer to driver */
```

```
        /*
         * Information required to be the parent of a disk wedge.
         */
        kmutex_t       dk_rawlock;    /* lock on these fields */
        u_int          dk_rawopens;   /* # of openes of rawvp */
        struct vnode   *dk_rawvp;     /* vnode for the RAW_PART bdev */

        kmutex_t       dk_openlock;   /* lock on these and openmask */
        u_int          dk_nwedges;    /* # of configured wedges */
                                      /* all wedges on this disk */
        LIST_HEAD(, dkwedge_softc) dk_wedges;

        /*
         * Disk label information.  Storage for the in-core disk label
         * must be dynamically allocated, otherwise the size of this
         * structure becomes machine-dependent.
         */
        daddr_t        dk_labelsector;               /* sector containing label */
        struct disklabel *dk_label;   /* label */
        struct cpu_disklabel *dk_cpulabel;
};
```

The system maintains a global linked-list of all disks attached to the system. This list, called **disklist**, may grow or shrink over time as disks are dynamically added and removed from the system. Drivers which currently make use of the detachment capability of the framework are the **ccd** and **vnd** pseudo-device drivers.

The following is a brief description of each function in the framework:

disk_init() Initialize the disk structure.

disk_attach() Attach a disk; allocate storage for the disklabel, set the "attached time" timestamp, insert the disk into the disklist, and increment the system disk count.

disk_detach() Detach a disk; free storage for the disklabel, remove the disk from the disklist, and decrement the system disk count. If the count drops below zero, panic.

disk_destroy() Release resources used by the disk structure when it is no longer required.

disk_busy() Increment the disk's "busy counter". If this counter goes from 0 to 1, set the timestamp corresponding to this transfer.

disk_unbusy() Decrement a disk's busy counter. If the count drops below zero, panic. Get the current time, subtract it from the disk's timestamp, and add the difference to the disk's running total. Set the disk's timestamp to the current time. If the provided byte count is greater than 0, add it to the disk's running total and increment the number of transfers performed by the disk. The third argument *read* specifies the direction of I/O; if non-zero it means reading from the disk, otherwise it means writing to the disk.

disk_find() Return a pointer to the disk structure corresponding to the name provided, or NULL if the disk does not exist.

disk_blocksize() Initialize *dk_blkshift* and *dk_byteshift* members of *struct disk* with suitable values derived from the supplied physical blocksize. It is only necessary to call this function if the device's physical blocksize is not DEV_BSIZE.

The functions typically called by device drivers are **disk_init**() **disk_attach**(), **disk_detach**(), **disk_destroy,**() **disk_busy**(), **disk_unbusy**(), and **disk_blocksize**(). The function **disk_find**() is provided as a utility function.

USING THE FRAMEWORK

This section includes a description on basic use of the framework and example usage of its functions. Actual implementation of a device driver which uses the framework may vary.

Each device in the system uses a "softc" structure which contains autoconfiguration and state information for that device. In the case of disks, the softc should also contain one instance of the disk structure, e.g.:

```
struct foo_softc {
        device_t        sc_dev;         /* generic device information */
        struct  disk    sc_dk;          /* generic disk information */
        [ . . . more . . . ]
};
```

In order for the system to gather metrics data about a disk, the disk must be registered with the system. The **disk_attach**() routine performs all of the functions currently required to register a disk with the system including allocation of disklabel storage space, recording of the time since boot that the disk was attached, and insertion into the disklist. Note that since this function allocates storage space for the disklabel, it must be called before the disklabel is read from the media or used in any other way. Before **disk_attach**() is called, a portions of the disk structure must be initialized with data specific to that disk. For example, in the "foo" disk driver, the following would be performed in the autoconfiguration "attach" routine:

```
void
fooattach(device_t parent, device_t self, void *aux)
{
        struct foo_softc *sc = device_private(self);
        [ . . . ]

        /* Initialize and attach the disk structure. */
        disk_init(&sc->sc_dk, device_xname(self), &foodkdriver);
        disk_attach(&sc->sc_dk);

        /* Read geometry and fill in pertinent parts of disklabel. */
        [ . . . ]
        disk_blocksize(&sc->sc_dk, bytes_per_sector);
}
```

The **foodkdriver** above is the disk's "driver" switch. This switch currently includes a pointer to the disk's "strategy" routine. This switch needs to have global scope and should be initialized as follows:

```
void foostrategy(struct buf *);

const struct dkdriver foodkdriver = {
        .d_strategy = foostrategy,
};
```

Once the disk is attached, metrics may be gathered on that disk. In order to gather metrics data, the driver must tell the framework when the disk starts and stops operations. This functionality is provided by the **disk_busy**() and **disk_unbusy**() routines. The **disk_busy**() routine should be called immediately before a command to the disk is sent, e.g.:

```
void
foostart(sc)
        struct foo_softc *sc;
{
        [ . . . ]

        /* Get buffer from drive's transfer queue. */
        [ . . . ]

        /* Build command to send to drive. */
        [ . . . ]

        /* Tell the disk framework we're going busy. */
        disk_busy(&sc->sc_dk);

        /* Send command to the drive. */
        [ . . . ]
}
```

When **disk_busy**() is called, a timestamp is taken if the disk's busy counter moves from 0 to 1, indicating the disk has gone from an idle to non-idle state. Note that **disk_busy**() must be called at **splbio**(). At the end of a transaction, the **disk_unbusy**() routine should be called. This routine performs some consistency checks, such as ensuring that the calls to **disk_busy**() and **disk_unbusy**() are balanced. This routine also performs the actual metrics calculation. A timestamp is taken, and the difference from the timestamp taken in **disk_busy**() is added to the disk's total running time. The disk's timestamp is then updated in case there is more than one pending transfer on the disk. A byte count is also added to the disk's running total, and if greater than zero, the number of transfers the disk has performed is incremented. The third argument *read* specifies the direction of I/O; if non-zero it means reading from the disk, otherwise it means writing to the disk.

```
void
foodone(xfer)
        struct foo_xfer *xfer;
{
        struct foo_softc = (struct foo_softc *)xfer->xf_softc;
        struct buf *bp = xfer->xf_buf;
        long nbytes;
        [ . . . ]

        /*
         * Get number of bytes transfered.  If there is no buf
         * associated with the xfer, we are being called at the
         * end of a non-I/O command.
         */
        if (bp == NULL)
                nbytes = 0;
        else
                nbytes = bp->b_bcount - bp->b_resid;

        [ . . . ]

        /* Notify the disk framework that we've completed the transfer. */
        disk_unbusy(&sc->sc_dk, nbytes,
```

```
                bp != NULL ? bp->b_flags & B_READ : 0);

        [ . . . ]
}
```

Like **disk_busy**(), **disk_unbusy**() must be called at **splbio**().

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the disk framework can be found. All pathnames are relative to `/usr/src`.

The disk framework itself is implemented within the file `sys/kern/subr_disk.c`. Data structures and function prototypes for the framework are located in `sys/sys/disk.h`.

The NetBSD machine-independent SCSI disk and CD-ROM drivers use the disk framework. They are located in `sys/scsi/sd.c` and `sys/scsi/cd.c`.

The NetBSD **ccd** and **vnd** drivers use the detachment capability of the framework. They are located in `sys/dev/ccd.c` and `sys/dev/vnd.c`.

SEE ALSO

`ccd`(4), `vnd`(4), `spl`(9)

HISTORY

The NetBSD generic disk framework appeared in NetBSD 1.2.

AUTHORS

The NetBSD generic disk framework was architected and implemented by Jason R. Thorpe ⟨thorpej@NetBSD.org⟩.

NAME

disklabel, **readdisklabel**, **writedisklabel**, **setdisklabel**, **bounds_check_with_label** — disk label management routines

SYNOPSIS

```
char *
readdisklabel(dev_t dev, void (*strat)(struct buf *), struct disklabel *lp,
    struct cpu_disklabel *clp);

int
writedisklabel(dev_t dev, void (*strat)(struct buf *),
    struct disklabel *lp, struct cpu_disklabel *clp);

int
setdisklabel(struct disklabel *olp, struct disklabel *nlp, u_long openmask,
    struct cpu_disklabel *clp);

int
bounds_check_with_label(struct buf *bp, struct disklabel *lp, int wlabel);
```

DESCRIPTION

This collection of routines provides a disklabel management interface to kernel device drivers. These routines are classified as machine- or architecture-dependent because of restrictions imposed by the machine architecture and boot-strapping code on the location of the label, or because cooperation with other operating systems requires specialized conversion code.

readdisklabel() attempts to read a disklabel from the device identified by *dev*, using the device strategy routine passed in *strat*. Note that a buffer structure is required to pass to the strategy routine; it needs to be acquired and parameterized for the intended I/O operation, and disposed of when the operation has completed. Some fields in the disklabel passed in *lp* may be pre-initialized by the caller in order to meet device driver requirements for the I/O operation initiated to get to the disklabel data on the medium. In particular, the field "d_secsize", if non-zero, is used by **readdisklabel**() to get an appropriately sized buffer to pass to the device strategy routine. Unspecified fields in *lp* should be set to zero. If the medium does not contain a native disklabel that can be read in directly, **readdisklabel**() may resort to constructing a label from other machine-dependent information using the provided buffer passed in the *clp* argument. If a disk label can not be found or constructed, a string containing an approximated description of the failure mode is returned. Otherwise the NULL string is returned.

writedisklabel() stores disk label information contained in the disk label structure given by *lp* on the device identified by *dev*. Like **readdisklabel**(), it acquires and sets up an I/O buffer to pass to the strategy routine *strat*. **writedisklabel**() may elect to do a machine-dependent conversion of the native disk label structure (using the buffer pointed at by *clp*), to store the disk label onto the medium in a format complying with architectural constraints. **writedisklabel**() returns 0 on success and EINVAL if the disk label specifies invalid or inconvertible values. Otherwise, any error condition reported by the device strategy routine in the buffer's "*b_error*" field is returned.

setdisklabel() checks a proposed new disk label passed in *nlp* for some amount of basic sanity. This includes a check on attempts to change the location, or reduce the size, of an existing disk partition that is currently in use by the system. The current disposition of the disk partitions is made available through *olp* and *openmask*, which provide, respectively, the existing disk label and a bit mask identifying the partitions that are currently in use. Failure to pass on "basic sanity", results in a EINVAL return value, while a vetoed update of the partition layout is signaled by a EBUSY return value. Otherwise, 0 is returned.

bounds_check_with_label() is used to check whether a device transfer described by *bp* to the device identified by *dev*, is properly contained within a disk partition of the disk with label *lp*. If this check fails,

bounds_check_with_label() sets the buffer's "*b_error*" field to EINVAL, sets the B_ERROR flag in "*b_flags*", and returns -1. If the argument *wlabel* is zero, and the transfer is a write operation, a check is done if the transfer would overwrite (a portion of) the disklabel area on the medium. If that is the case, EROFS is set in "*b_error*", the B_ERROR flag is set in "*b_flags*", and -1 is returned. Note that *wlabel* should be set to a non-zero value if the intended operation is expected to install or update the disk label. Programs that intend to do so using the raw device interface should notify the driver by using a DIOCWLABEL ioctl function.

SEE ALSO

disklabel(5), disklabel(8)

NAME

dmover_backend_register, **dmover_backend_unregister**, **dmover_session_create**, **dmover_session_destroy**, **dmover_request_alloc**, **dmover_request_free**, **dmover_process**, **dmover_done** — hardware-assisted data mover interface

SYNOPSIS

```
#include <dev/dmover/dmovervar.h>
```

Client interface routines:

int
dmover_session_create(*const char* *, *struct dmover_session* **);

void
dmover_session_destroy(*struct dmover_session* *);

struct dmover_request *
dmover_request_alloc(*struct dmover_session* *, *dmover_buffer* *);

void
dmover_request_free(*struct dmover_request* *);

void
dmover_process(*struct dmover_request* *);

Back-end interface routines:

void
dmover_backend_register(*struct dmover_backend* *);

void
dmover_backend_unregister(*struct dmover_backend* *);

void
dmover_done(*struct dmover_request* *);

DESCRIPTION

The **dmover** facility provides an interface to hardware-assisted data movers. This can be used to copy data from one location in memory to another, clear a region of memory, fill a region of memory with a pattern, and perform simple operations on multiple regions of memory, such as an XOR, without intervention by the CPU.

The drivers for hardware-assisted data movers present themselves to **dmover** by registering their capabilities. When a client wishes to use a **dmover** function, it creates a session for that function, which identifies back-ends capable of performing that function. The client then enqueues requests on that session, which the back-ends process asynchronously. The client may choose to block until the request is completed, or may have a call-back invoked once the request has been completed.

When a client creates a session, the **dmover** facility identifies back-ends which are capable of handling the requested function. When a request is scheduled for processing, the **dmover** scheduler will identify the best back-end to process the request from the list of candidate back-ends, in an effort to provide load balancing, while considering the relative performance of each back-end.

A **dmover** function always has one output region. A function may have zero or more input regions, or may use an immediate value as an input. For functions which use input regions, the lengths of each input region and the output region must be the same. All **dmover** functions with the same name will have the same number of and type inputs. If a back-end attempts to register a function which violates this invariant, behavior is undefined.

The **dmover** facility supports several types of buffer descriptors. For functions which use input regions, each input buffer descriptor and the output buffer descriptor must be of the same type. This restriction may be removed in a future revision of the interface.

The **dmover** facility may need to interrupt request processing and restart it. Clients of the **dmover** facility should take care to avoid unwanted side-effects should this occur. In particular, for functions which use input regions, no input region may overlap with the output region.

DATA STRUCTURES

The **dmover** facility shares several data structures between the client and back-end in order to describe sessions and requests.

```
typedef enum {
        DMOVER_BUF_LINEAR,
        DMOVER_BUF_UIO
} dmover_buffer_type;

typedef struct {
        void *l_addr;
        size_t l_len;
} dmover_buf_linear;

typedef union {
        dmover_buf_linear dmbuf_linear;
        struct uio *dmbuf_uio;
} dmover_buffer;
```

Together, these data types are used to describe buffer data structures which the **dmover** facility understands. Additional buffer types may be added in future revisions of the **dmover** interface.

The *dmover_assignment* structure contains the information about the back-end to which a request is currently assigned. It contains the following public members:

struct dmover_backend *das_backend
: This is a pointer to the back-end.

const struct dmover_algdesc *das_algdesc
: This is a pointer to the algorithm description provided by the back-end for the request's function.

The *dmover_session* structure contains the following public members:

void *dses_cookie
: This is a pointer to client private data.

int dses_ninputs
: This is the number of inputs used by the selected function.

The *dmover_request* structure contains the following public members:

TAILQ_ENTRY(dmover_request) dreq_dmbq
: Linkage on the back-end's queue of pending requests.

struct dmover_session *dreq_session
: Pointer to the session with which this request is associated. This is intended for use by the back-end.

struct dmover_assignment *dreq_assignment
: Pointer to the *dmover_assignment* structure which describes the back-end to which the request is currently assigned. The back-end is assigned when the request is scheduled with

dmover_process().

void (*dreq_callback)(struct dmover_request *)
: This is a pointer to an optional call-back function provided by the client. If provided, the call-back is invoked when the request is complete. This field must be `NULL` if *DMOVER_REQ_WAIT* is set in *dreq_flags*.

void *dreq_cookie
: This is a pointer to client private data specific to the request.

void *dreq_dmbcookie
: This is a pointer to back-end private data, for use while the back-end is actively processing a request.

volatile int dreq_flags
: The following flags are defined:

DMOVER_REQ_DONE	The request has been completed. If not using a call-back, the client may poll this bit to determine if a request has been processed.
DMOVER_REQ_ERROR	An error has occurred while processing the request.
DMOVER_REQ_RUNNING	The request is currently being executed by the back-end. Once a command is running, it cannot be cancelled, and must run to completion.
DMOVER_REQ_WAIT	If set by the client, **dmover_process**() will wait for the request to complete using `cv_wait`(9). This flag may only be used if the caller has a valid thread context. If this flag is set, a callback may not be used.

int dreq_error
: If the *DMOVER_REQ_ERROR* bit is set, this contains the `errno`(2) value indicating the error that occurred during processing.

dmover_buffer_type dreq_outbuf_type
: The type of the output buffer.

dmover_buffer dreq_outbuf
: The output buffer.

uint8_t dreq_immediate[8]
: This is the input for algorithms which use an immediate value. Values smaller than 8 bytes should use the least-significant bytes first. For example, a 32-bit integer would occupy bytes 0, 1, 2, and 3.

dmover_buffer_type dreq_inbuf_type
: The type of the input buffer. This is only used if the **dmover** function has one or more inputs.

dmover_buffer *dreq_inbuf
: A pointer to an array of input buffers. This is only used if the **dmover** function has one or more inputs. The number of inputs, and thus the number of valid elements in the array, is specified by the algorithm description for the session.

CLIENT INTERFACE

The following functions are provided to the client:

int **dmover_session_create**(*const char *function*, *struct dmover_session **sessionp*)

The **dmover_session_create**() function creates a data mover session for the specified data movement function *function*. A handle to the new session is returned in **sessionp*.

The following are valid data movement function names:

"zero" Fill a memory region with zeros. This algorithm has an input count of 0.

"fill8" Fill a memory region with an 8-bit pattern. This algorithm has an input count of 0. The pattern is provided in the *dreq_imm8* member of the *dmover_request* structure.

"copy" Copy a memory region from one location to another. This algorithm has an input count of 1.

"xor2" Perform an XOR operation on 2 inputs. This algorithm has an input count of 2.

"xor3" Perform an XOR operation on 3 inputs. This algorithm has an input count of 3.

"xor4" Perform an XOR operation on 4 inputs. This algorithm has an input count of 4.

"xor5" Perform an XOR operation on 5 inputs. This algorithm has an input count of 5.

"xor6" Perform an XOR operation on 6 inputs. This algorithm has an input count of 6.

"xor7" Perform an XOR operation on 7 inputs. This algorithm has an input count of 7.

"xor8" Perform an XOR operation on 8 inputs. This algorithm has an input count of 8.

Users of the **dmover** facility are encouraged to use the following aliases for the well-known function names, as doing so saves space and reduces the chance of programming errors:

DMOVER_FUNC_ZERO "zero" (*dmover_funcname_zero*)

DMOVER_FUNC_FILL8 "fill8" (*dmover_funcname_fill8*)

DMOVER_FUNC_COPY "copy" (*dmover_funcname_copy*)

DMOVER_FUNC_XOR2 "xor2" (*dmover_funcname_xor2*)

DMOVER_FUNC_XOR3 "xor3" (*dmover_funcname_xor3*)

DMOVER_FUNC_XOR4 "xor4" (*dmover_funcname_xor4*)

DMOVER_FUNC_XOR5 "xor5" (*dmover_funcname_xor5*)

DMOVER_FUNC_XOR6 "xor6" (*dmover_funcname_xor6*)

DMOVER_FUNC_XOR7 "xor7" (*dmover_funcname_xor7*)

DMOVER_FUNC_XOR8 "xor8" (*dmover_funcname_xor8*)

void **dmover_session_destroy**(*struct dmover_session *session*)

The **dmover_session_destroy**() function tears down a data mover session and releases all resources associated with it.

struct dmover_request * **dmover_request_alloc**(*struct dmover_session *session*, *dmover_buffer *inbuf*)

The **dmover_request_alloc**() function allocates a **dmover** request structure and associates it with the specified session. If the *inbuf* argument is not NULL, *inbuf* is used as the array of input buffer descriptors in the request. Otherwise, if *inbuf* is NULL and the **dmover** function requires input buffers, the input buffer descriptors will be allocated automatically using malloc(9).

If the request structure or input buffer descriptors cannot be allocated, **dmover_request_alloc**() return NULL to indicate failure.

void **dmover_request_free**(*struct dmover_request *req*)

The **dmover_request_free**() function frees a **dmover** request structure. If the **dmover** function requires input buffers, and the input buffer descriptors associated with *req* were allocated by **dmover_request_alloc**(), then the input buffer descriptors will also be freed.

void **dmover_process**(*struct dmover_request *req*)

The **dmover_process**() function submits the **dmover** request *req* for processing. The call-back specified by the request is invoked when processing is complete.

The **dmover_session_create**() and **dmover_session_destroy**() functions must not be called from interrupt context.

The **dmover_request_alloc**(), **dmover_request_free**(), and **dmover_process**() functions may be called from interrupt handlers at levels *IPL_VM*, *IPL_SOFTCLOCK*, and *IPL_SOFTNET*, or in non-interrupt context.

The request completion call-back is called from a software interrupt handler at *IPL_SOFTCLOCK*.

BACK-END INTERFACE

A back-end describes the **dmover** functions it can perform using an array of *dmover_algdesc* structures:

```
struct dmover_algdesc {
        const char *dad_name;  /* algorithm name */
        void *dad_data;        /* opaque algorithm description */
        int dad_ninputs;       /* number of inputs */
};
```

The *dad_name* member points to a valid **dmover** function name which the client may specify. The *dad_data* member points to a back-end-specific description of the algorithm.

A back-end presents itself to the **dmover** facility using the *dmover_backend* structure. The back-end must initialize the following members of the structure:

const char *dmb_name
: This is the name of the back-end.

u_int dmb_speed
: This is an estimate of the number of kilobytes/second that the back-end can process.

void *dmb_cookie
: This is a pointer to back-end private data.

const struct dmover_algdesc *dmb_algdescs
: This points to an array of *dmover_algdesc* structures which describe the functions the data mover can perform.

int dmb_nalgdescs
: This is the number of elements in the *dmb_algdescs* array.

void (*dmb_process)(struct dmover_backend *)
: This is the entry point to the back-end used to process requests.

When invoked by the **dmover** facility, the back-end's **(*dmb_process)**() function should examine the pending request queue in its *dmover_backend* structure:

TAILQ_HEAD(, dmover_request) dmb_pendreqs
This is the queue of pending requests.

int dmb_npendreqs
This is the number of requests in the *dmb_pendreqs* queue.

If an error occurs when processing the request, the *DMOVER_REQ_ERROR* bit must be set in the *dreq_flags* member of the request, and the *dreq_error* member set to an `errno`(2) value to indicate the error.

When the back-end has finished processing the request, it must call the **dmover_done**() function. This function eventually invokes the client's call-back routine.

If a hardware-assisted data mover uses interrupts, the interrupt handlers should be registered at IPL_VM.

The following functions are provided to the back-ends:

void **dmover_backend_register**(*struct dmover_backend *backend*)

The **dmover_backend_register**() function registers the back-end *backend* with the **dmover** facility.

void **dmover_backend_unregister**(*struct dmover_backend *backend*)

The **dmover_backend_unregister**() function removes the back-end *backend* from the **dmover** facility. The back-end must already be registered.

void **dmover_done**(*struct dmover_request *req*)

The **dmover_done**() function is called by the back-end when it has finished processing a request, whether the request completed successfully or not.

The **dmover_backend_register**() and **dmover_backend_unregister**() functions must not be called from interrupt context.

The **dmover_done**() function may be called at *IPL_VM*, *IPL_SOFTCLOCK*, *IPL_SOFTNET*, or in non-interrupt context.

EXAMPLES

The following is an example of a client using **dmover** to zero-fill a region of memory. In this example, the CPU will be able to context switch to another thread and perform work while the hardware-assisted data mover clears the specified block of memory.

```
int
hw_bzero(void *buf, size_t len)
{
        struct dmover_session *dses;
        struct dmover_request *dreq;
        int error;

        error = dmover_session_create(DMOVER_FUNC_ZERO, &dses);
        if (error)
                return (error);

        dreq = dmover_request_alloc(dses, NULL);
        if (dreq == NULL) {
                dmover_session_destroy(dses);
                return (ENOMEM);
        }
```

```
        dreq->dreq_flags = DMOVER_REQ_WAIT;
        dreq->dreq_callback = NULL;
        dreq->dreq_outbuf.dreq_outbuf_type = DMOVER_BUF_LINEAR;
        dreq->dreq_outbuf.dmbuf_linear.l_addr = buf;
        dreq->dreq_outbuf.dmbuf_linear.l_len = len;

        dmover_process(dreq);

        error = (dreq->dreq_flags & DMOVER_REQ_ERROR) ?
            dreq->dreq_error : 0;

        dmover_request_free(dreq);
        dmover_session_destroy(dses);

        return (error);
}
```

SEE ALSO

queue(3), dmoverio(4)

HISTORY

The **dmover** facility first appeared in NetBSD 2.0.

AUTHORS

The **dmover** facility was designed and implemented by Jason R. Thorpe ⟨thorpej@wasabisystems.com⟩ and contributed by Wasabi Systems, Inc.

BUGS

The mechanism by which a back-end should advertise its performance to the request scheduler is not well-defined. Therefore, the load-balancing mechanism within the request scheduler is also not well-defined.

NAME

do_setresuid, **do_setresgid** — set process uid and gid

SYNOPSIS

```
#include <sys/ucred.h>

int
do_setresuid(struct lwp *lwp, uid_t ruid, uid_t euid, uid_t svuid,
    u_int flags);

int
do_setresgid(struct lwp *lwp, uid_t ruid, uid_t euid, uid_t svuid,
    u_int flags);
```

DESCRIPTION

The **do_setresuid** and **do_setresgid** functions are used to implement the various system calls that allow a process to change its real, effective, and saved uid and gid values.

The **do_setresuid** function sets the specified processes real user ID to *ruid*, its effective user ID to *euid*, and its saved user ID to *svuid*. If any of the uid arguments are −1 then that assignment is skipped.

If **suser**() is true, then any values may be assigned, otherwise the new uid values must match one of the existing values and the caller must have set the relevant bit in *flags*.

The *flags* argument specifies which of the existing uid values the new value must match. It should be set to a logical OR of ID_{R,E,S}_EQ_{R,E,S}, where ID_E_EQ_R means that it is valid to set the effective ID to the current value of the real ID.

The **do_setresgid** function sets the group IDs but otherwise behaves in the same manner as **do_setresuid**. The processes group list is neither examined nor effected.

SEE ALSO

setregid(2), setreuid(2), setuid(2), suser(9)

CODE REFERENCES

These functions are implemented in: sys/kern/kern_prot.c.

HISTORY

Implemented for NetBSD 2.0 to replace ad-hoc code in each system call routine and in the various compat modules.

NAME

dofileread, **dofilereadv**, **dofilewrite**, **dofilewritev** — high-level file operations

SYNOPSIS

```
#include <sys/file.h>

int
dofileread(struct lwp *l, int fd, struct file *fp, void *buf, size_t nbyte,
    off_t *offset, int flags, register_t *retval);

int
dofilewrite(struct lwp *l, int fd, struct file *fp, const void *buf,
    size_t nbyte, off_t *offset, int flags, register_t *retval);

int
dofilereadv(struct lwp *l, int fd, struct file *fp,
    const struct iovec *iovp, int iovcnt, off_t *offset, int flags,
    register_t *retval);

int
dofilewritev(struct lwp *l, int fd, struct file *fp,
    const struct iovec *iovp, int iovcnt, off_t *offset, int flags,
    register_t *retval);
```

DESCRIPTION

The functions implement the underlying functionality of the read(2), write(2), readv(2), and writev(2) system calls. They are also used throughout the kernel as high-level access routines for file I/O.

The **dofileread**() function attempts to read *nbytes* of data from the object referenced by file entry *fp* into the buffer pointed to by *buf*. The **dofilewrite**() function attempts to write *nbytes* of data to the object referenced by file entry *fp* from the buffer pointed to by *buf*.

The **dofilereadv**() and **dofilewritev**() functions perform the same operations, but scatter the data with the *iovcnt* buffers specified by the members of the *iov* array.

The offset of the file operations is explicitly specified by **offset*. The new file offset after the file operation is returned in **offset*. If the FOF_UPDATE_OFFSET flag is specified in the *flags* argument, the file offset in the file entry *fp* is updated to reflect the new file offset, otherwise it remains unchanged after the operation.

The file descriptor *fd* is largely unused except for use by the ktrace framework for reporting to userlevel the process's file descriptor.

Upon successful completion the number of bytes which were transferred is returned in **retval*.

RETURN VALUES

Upon successful completion zero is returned, otherwise an appropriate error is returned.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using these file operations can be found. All pathnames are relative to /usr/src.

The framework for these file operations is implemented within the file sys/kern/sys_generic.c.

SEE ALSO

`file`(9)

NAME

dopowerhooks — run all power hooks

SYNOPSIS

void
dopowerhooks(*int why*);

DESCRIPTION

The **dopowerhooks**() function invokes all power hooks established using the powerhook_establish(9) function. When power is disappearing the power hooks are called in reverse order, i.e., the power hook established last will be called first. When power is restored they are called normal order.

This function is called from the apm(4) driver when a power change is detected.

SEE ALSO

powerhook_establish(9)

NAME

doshutdownhooks — run all shutdown hooks

SYNOPSIS

void
doshutdownhooks(*void*);

DESCRIPTION

The **doshutdownhooks**() function invokes all shutdown hooks established using the shutdownhook_establish(9) function. Shutdown hooks are called in reverse order, i.e., the shutdown hook established last will be called first.

This function is called from **cpu_reboot**() with interrupts turned off. It is called immediately before the system is halted or rebooted, after file systems have been unmounted, after the clock has been updated, and after a system dump has been done (if necessary).

SEE ALSO

cpu_reboot(9), shutdownhook_establish(9)

NAME

driver — description of a device driver

SYNOPSIS

```
#include <sys/param.h>
#include <sys/device.h>
#include <sys/errno.h>

static int
foo_match(struct device *parent, struct cfdata *match, void *aux);

static void
foo_attach(struct device *parent, struct device *self, void *aux);

static int
foo_detach(struct device *self, int flags);

static int
foo_activate(struct device *self, enum devact act);
```

DESCRIPTION

This page briefly describes the basic NetBSD autoconfiguration interface used by device drivers. For a detailed overview of the autoconfiguration framework see `autoconf`(9).

Each device driver must present to the system a standard autoconfiguration interface. This interface is provided by the *cfattach* structure. The interface to the driver is constant and is defined statically inside the driver. For example, the interface to driver "foo" is defined with:

```
CFATTACH_DECL(foo,                      /* driver name */
        sizeof(struct foo_softc),       /* size of instance data */
        foo_match,                      /* match/probe function */
        foo_attach,                     /* attach function */
        foo_detach,                     /* detach function */
        foo_activate);                  /* activate function */
```

For each device instance controlled by the driver, the autoconfiguration framework allocates a block of memory to record device-instance-specific driver variables. The size of this memory block is specified by the second argument in the *CFATTACH_DECL* macro. The memory block is referred to as the driver's *softc* structure. The *softc* structure is only accessed within the driver, so its definition is local to the driver. Nevertheless, the *softc* structure should adopt the standard NetBSD configuration and naming conventions. For example, the *softc* structure for driver "foo" is defined with:

```
struct foo_softc {
        struct device sc_dev;           /* generic device info */
        /* device-specific state */
};
```

The autoconfiguration framework mandates that the first member of the *softc* structure must be the driver-independent *struct device*. Probably its most useful aspect to the driver is that it contains the device-instance name *dv_xname*.

If a driver has character device interfaces accessed from userland, the driver must define the *cdevsw* structure. The structure is constant and is defined inside the driver. For example, the *cdevsw* structure for driver "foo" is defined with:

```
const struct cdevsw foo_cdevsw {
        int (*d_open)(dev_t, int, int, struct lwp *);
```

```
        int (*d_close)(dev_t, int, int, struct lwp *);
        int (*d_read)(dev_t, struct uio *, int);
        int (*d_write)(dev_t, struct uio *, int);
        int (*d_ioctl)(dev_t, u_long, void *, int, struct lwp *);
        void (*d_stop)(struct tty *, int);
        struct tty *(*d_tty)(dev_t);
        int (*d_poll)(dev_t, int, struct lwp *);
        paddr_t (*d_mmap)(dev_t, off_t, int);
        int (*d_kqfilter)(dev_t, struct knote *);
        int d_type;
};
```

The structure variable must be named foo_cdevsw by appending the letters "_cdevsw" to the driver's base name. This convention is mandated by the autoconfiguration framework.

If the driver "foo" has also block device interfaces, the driver must define the *bdevsw* structure. The structure is constant and is defined inside the driver. For example, the *bdevsw* structure for driver "foo" is defined with:

```
const struct bdevsw foo_bdevsw {
        int (*d_open)(dev_t, int, int, struct lwp *);
        int (*d_close)(dev_t, int, int, struct lwp *);
        void (*d_strategy)(struct buf *);
        int (*d_ioctl)(dev_t, u_long, void *, int, struct lwp *);
        int (*d_dump)(dev_t, daddr_t, void *, size_t);
        int (*d_psize)(dev_t);
        int d_type;
};
```

The structure variable must be named foo_bdevsw by appending the letters "_bdevsw" to the driver's base name. This convention is mandated by the autoconfiguration framework.

During system bootstrap, the autoconfiguration framework searches the system for devices. For each device driver, its match function is called (via its *cfattach* structure) to match the driver with a device instance. The match function is called with three arguments. This first argument *parent* is a pointer to the driver's parent device structure. The second argument *match* is a pointer to a data structure describing the autoconfiguration framework's understanding of the driver. Both the *parent* and *match* arguments are ignored by most drivers. The third argument *aux* contains a pointer to a structure describing a potential device-instance. It is passed to the driver from the parent. The match function would type-cast the *aux* argument to its appropriate attachment structure and use its contents to determine whether it supports the device. Depending on the device hardware, the contents of the attachment structure may contain "locators" to locate the device instance so that the driver can probe it for its identity. If the probe process identifies additional device properties, it may modify the members of the attachment structure. For these devices, the NetBSD convention is to call the match routine **foo_probe**() instead of **foo_match**() to make this distinction clear. Either way, the match function returns a nonzero integer indicating the confidence of supporting this device and a value of 0 if the driver doesn't support the device. Generally, only a single driver exists for a device, so the match function returns 1 for a positive match.

The autoconfiguration framework will call the attach function (via its *cfattach* structure) of the driver which returns the highest value from its match function. The attach function is called with three arguments. The attach function performs the necessary process to initialise the device for operation. The first argument *parent* is a pointer to the driver's parent device structure. The second argument *self* is a pointer to the driver's device structure. It is also a pointer to our *softc* structure since the device structure is its first member. The third argument *aux* is a pointer to the attachment structure. The *parent* and *aux* arguments are the same as passed to the match function.

The driver's attach function is called before system interrupts are enabled. If interrupts are required during initialisation, then the attach function should make use of **config_interrupts**() (see autoconf(9)).

Some devices can be removed from the system without requiring a system reboot. The autoconfiguration framework calls the driver's detach function (via its *cfattach* structure) during device detachment. If the device does not support detachment, then the driver does not have to provide a detach function. The detach function is used to relinquish resources allocated to the driver which are no longer needed. The first argument *self* is a pointer to the driver's device structure. It is the same structure as passed to the attach function. The second argument *flags* contains detachment flags. Valid values are DETACH_FORCE (force detachment; hardware gone) and DETACH_QUIET (do not print a notice).

The autoconfiguration framework may call the driver's activate function to notify the driver of a change in the resources that have been allocated to it. For example, an Ethernet driver has to be notified if the network stack is being added or removed from the kernel. The first argument to the activate function *self* is a pointer to the driver's device structure. It is the same argument as passed to the attach function. The second argument *act* describes the action. Valid actions are DVACT_ACTIVATE (activate the device) and DVACT_DEACTIVATE (deactivate the device). If the action is not supported the activate function should return EOPNOTSUPP. The DVACT_DEACTIVATE call will only be made if the DVACT_ACTIVATE call was successful. The activate function is called in interrupt context.

Most drivers will want to make use of interrupt facilities. Interrupt locators provided through the attachment structure should be used to establish interrupts within the system. Generally, an interrupt interface is provided by the parent. The interface will require a handler and a driver-specific argument to be specified. This argument is usually a pointer to the device-instance-specific softc structure. When a hardware interrupt for the device occurs the handler is called with the argument. Interrupt handlers should return 0 for "interrupt not for me", 1 for "I took care of it", or -1 for "I guess it was mine, but I wasn't expecting it".

For a driver to be compiled into the kernel, config(1) must be aware of its existence. This is done by including an entry in files.<bus> in the directory containing the driver. For example, the driver "foo" attaching to bus "bar" with dependency on kernel module "baz" has the entry:

```
device  foo: baz
attach  foo at bar
file    dev/bar/foo.c           foo
```

An entry can now be added to the machine description file:

```
foo*    at bar?
```

For device interfaces of a driver to be compiled into the kernel, config(1) must be aware of its existence. This is done by including an entry in majors.<arch>. For example, the driver "foo" with character device interfaces, a character major device number "cmaj", block device interfaces, a block device major number "bmaj" and dependency on kernel module "baz" has the entry:

```
device-major    foo     char cmaj block bmaj   baz
```

For a detailed description of the machine description file and the "device definition" language see config(9).

SEE ALSO

config(1), autoconf(9), config(9), powerhook_establish(9), shutdownhook_establish(9)

NAME

edid — VESA Extended Display Identification Data

SYNOPSIS

```
#include <dev/videomode/edidvar.h>
#include <dev/videomode/edidreg.h>

int
edid_is_valid(uint8_t *data);

int
edid_parse(uint8_t *data, struct edid_info *info);

void
edid_print(struct edid_info *info);
```

DESCRIPTION

These functions provide support parsing the Extended Display Identification Data which describes a display device such as a monitor or flat panel display.

The **edid_is_valid**() function simply tests if the EDID block in *data* contains valid data. This test includes a verification of the checksum, and that valid vendor and product idenfication data is present. The data block contain at least 128 bytes.

The **edid_parse**() function parses the supplied *data* block (which again, must be at least 128 bytes), writing the relevant data into the structure pointed to by *info*.

The **edid_print**() function prints the data in the given *info* structure to the console device.

RETURN VALUES

The **edid_is_valid**() function returns 0 if the data block is valid, and EINVAL otherwise. The **edid_parse**() function returns zero if the data was successfully parsed, and non-zero otherwise.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the EDID subsystem can be found. All pathnames are relative to /usr/src.

The EDID subsystem is implemented within the file sys/dev/videomode/edid.c.

The EDID subsystem also makes use of VESA Generalized Timing Formula located located in sys/dev/videomode/vesagtf.c and the generic videomode database located in sys/dev/videomode/videomode.c.

EXAMPLES

The following code uses these functions to retrieve and print information about a monitor:

```
struct edid_info info;
i2c_tag_t        tag;
char            buffer[128];

...
/* initialize i2c tag... */
...
if ((ddc_read_edid(tag, buffer, 128) == 0) &&
    (edid_parse(buffer, &info) == 0))
        edid_print(info);
...
```

SEE ALSO

`ddc`(9), `edid`(9), `iic`(9)

HISTORY

These routines were added in NetBSD 4.0.

AUTHORS

Garrett D'Amore ⟨gdamore@NetBSD.org⟩

NAME

errno — kernel internal error numbers

SYNOPSIS

```
#include <sys/errno.h>
```

DESCRIPTION

This section provides an overview of the error numbers used internally by the kernel and indicate neither success nor failure. These error numbers are not returned to userland code.

DIAGNOSTICS

Kernel functions that indicate success or failure by means of either 0 or an errno(2) value sometimes have a need to indicate that "special" handling is required at an upper layer or, in the case of ioctl(2) processing, that "nothing was wrong but the request was not handled". To handle these cases, some negative errno(2) values are defined which are handled by the kernel before returning a different errno(2) value to userland or simply zero.

The following is a list of the defined names and their meanings as given in ⟨errno.h⟩. It is important to note that the value −1 is *not* used, since it is commonly used to indicate generic failure and leaves it up to the caller to determine the action to take.

−2 EJUSTRETURN *Modify regs, just return*. No more work is required and the function should just return.

−3 ERESTART *Restart syscall*. The system call should be restarted. This typically means that the machine dependent system call trap code will reposition the process's instruction pointer or program counter to re-execute the current system call with no other work required.

−4 EPASSTHROUGH *Operation not handled by this layer*. The operation was not handled and should be passed through to another layer. This often occurs when processing ioctl(2) requests since lower layer processing may not handle something that subsequent code at a higher level will.

−5 EDUPFD *Duplicate file descriptor.* This error is returned from the device open routine indicating that the *l_dupfd* field contains the file descriptor information to be returned to the caller, instead of the file descriptor that has been opened already. This error is used by cloning device multiplexors. Cloning device multiplexors open a new file descriptor and associate that file descriptor with the appropriate cloned device. They set *l_dupfd* to that new file descriptor and return EDUPFD. vn_open(9) takes the file descriptor pointed to by *l_dupfd* and copies it to the file descriptor that the open call will return.

−6 EMOVEFD *Move file descriptor.* This error is similar to EDUPFD except that the file descriptor in *l_dupfd* is closed after it has been copied.

SEE ALSO

errno(2), ioctl(9)

HISTORY

An **errno** manual page appeared in Version 6 AT&T UNIX. This **errno** manual page appeared in NetBSD 3.0.

NAME

ethersubr, **ether_ifattach**, **ether_addmulti**, **ether_delmulti**, **ETHER_FIRST_MULTI**, **ETHER_NEXT_MULTI**, **ETHER_IS_MULTICAST**, **fddi_ifattach**, **fddi_addmulti**, **fddi_delmulti** — Ethernet and FDDI driver support functions and macros

SYNOPSIS

```
#include <net/if_ether.h>

void
ether_ifattach(struct ifnet *ifp, uint8_t *lla);

int
ether_addmulti(const struct sockaddr *sa, struct ethercom *ec);

int
ether_delmulti(const struct sockaddr *sa, struct ethercom *ec);

void
ETHER_FIRST_MULTI(struct ether_multistep step, struct ethercom *ec,
    struct ether_multi *enm);

void
ETHER_NEXT_MULTI(struct ether_multistep step, struct ether_multi *enm);

int
ETHER_IS_MULTICAST(uint8_t *addr);

#include <net/if_fddi.h>

void
fddi_ifattach(struct ifnet *ifp, uint8_t *lla);

int
fddi_addmulti(const struct sockaddr *sa, struct ethercom *ec);

int
fddi_delmulti(const struct sockaddr *sa, struct ethercom *ec);
```

DESCRIPTION

The **ethersubr** functions provide the interface between the **ethersubr** module and the network drivers which need Ethernet support. Such drivers must request the *ether* attribute in their *files* declaration and call the appropriate functions as specified below.

FDDI drivers must request the "fddi" attribute in their "files" declaration and call the functions tagged with "fddi_" or "FDDI_" instead, where different. Some macros are shared.

Note that you also need the arp(9) stuff to support IPv4 on your hardware.

ether_ifattach()
: Perform the device-independent, but Ethernet-specific initialization of the interface pointed to by *ifp*.

  Among other duties, this function creates a record for the link level address in the interface's address list and records the link level address pointed to by *lla* there.

  You must call this function from the driver's attach function.

fddi_ifattach()
: corresponding function for FDDI devices.

ether_addmulti()

ether_delmulti()
Add (**ether_addmulti**()) or delete (**ether_delmulti**()) the address described by the *sa* pointer to the Ethernet multicast list belonging to *ec*.

These functions must be called from the driver's ioctl function to handle SIOCADDMULTI and SIOCDELMULTI requests. If they return ENETRESET, the hardware multicast filter must be reinitialized.

These functions accept AF_UNSPEC addresses, which are interpreted as Ethernet addresses, or AF_INET addresses. In the latter case, INADDR_ANY is mapped to a range describing all the Ethernet address space reserved for IPv4 multicast addresses.

ether_addmulti() returns EAFNOSUPPORT if an unsupported address family is specified, EINVAL if a non-multicast address is specified, or ENETRESET if the multicast list really changed and the driver should synchronize its hardware filter with it.

ether_delmulti() returns, in addition to the above errors, ENXIO if the specified address can't be found in the list of multicast addresses.

fddi_addmulti()

fddi_delmulti()
corresponding functions for FDDI devices.

ETHER_NEXT_MULTI()
is a macro to step through all of the ether_multi records, one at a time. The current position is remembered in *step*, which the caller must provide.

ETHER_FIRST_MULTI()
must be called to initialize *step* and get the first record. Both macros return a NULL *enm* when there are no remaining records.

ETHER_IS_MULTICAST()
returns 1, if *addr* points to an Ethernet/FDDI multicast (or broadcast) address. Implemented as a macro.

SEE ALSO
arp(9)

AUTHORS
UCB CSRG (original implementation)
Ignatios Souvatzis (support for new ARP system)

CODE REFERENCES
Ethernet support functions are declared in ⟨net/if_ether.h⟩ and defined (if not implemented as macro) in /usr/src/sys/net/if_ethersubr.c.

FDDI support functions are declared in ⟨net/if_fddi.h⟩ and defined (if not implemented as macro) in /usr/src/sys/net/if_fddisubr.c.

HISTORY
Rewritten to attach to the new ARP system in NetBSD 1.3.

NAME

evcnt, **evcnt_attach_dynamic**, **evcnt_attach_static**, **evcnt_detach** — generic event counter framework

SYNOPSIS

```
#include <sys/evcnt.h>

void
evcnt_attach_dynamic(struct evcnt *ev, int type,
    const struct evcnt *parent, const char *group, const char *name);

void
evcnt_attach_static(struct evcnt *ev);

void
evcnt_detach(struct evcnt *ev);
```

DESCRIPTION

The NetBSD generic event counter framework is designed to provide a flexible and hierarchical event counting facility, which is useful for tracking system events (including device interrupts).

The fundamental component of this framework is the **evcnt** structure. Its user-accessible fields are:

```
struct evcnt {
        uint64_t        ev_count;       /* how many have occurred */
        TAILQ_ENTRY(evcnt) ev_list;     /* entry on list of all counters */
        unsigned char   ev_type;        /* counter type; see below */
        unsigned char   ev_grouplen;    /* 'group' len, excluding NUL */
        unsigned char   ev_namelen;     /* 'name' len, excluding NUL */
        const struct evcnt *ev_parent;/* parent, for hierarchical ctrs */
        const char      *ev_group;      /* name of group */
        const char      *ev_name;       /* name of specific event */
};
```

The system maintains a global linked list of all active event counters. This list, called **allevents**, may grow or shrink over time as event counters are dynamically added to and removed from the system.

Each event counter is marked (in the *ev_type* field) with the type of event being counted. The following types are currently defined:

- `EVCNT_TYPE_MISC` Miscellaneous; doesn't fit into one of the other types.
- `EVCNT_TYPE_INTR` Interrupt counter, reported by **vmstat -i**.
- `EVCNT_TYPE_TRAP` Processor trap style events.

Each event counter also has a group name (*ev_group*) and an event name (*ev_name*) which are used to identify the counter. The group name may be shared by a set of counters. For example, device interrupt counters would use the name of the device whose interrupts are being counted as the group name. The counter name is meant to distinguish the counter from others in its group (and need not be unique across groups). Both names should be understandable by users, since they are printed by commands like `vmstat`(1). The constant `EVCNT_STRING_MAX` is defined to be the maximum group or event name length in bytes (including the trailing `NUL`). In the current implementation it is 256.

To support hierarchical tracking of events, each event counter can name a "parent" event counter. For instance, interrupt dispatch code could have an event counter per interrupt line, and devices could each have counters for the number of interrupts that they were responsible for causing. In that case, the counter for a device on a given interrupt line would have the line's counter as its parent. The value `NULL` is used to indicate that a counter has no parent. A counter's parent must be attached before the counter is attached, and

detached after the counter is detached.

The **EVCNT_INITIALIZER**() macro can be used to provide a static initializer for an event counter structure. It is invoked as **EVCNT_INITIALIZER**(*type*, *parent*, *group*, *name*), and its arguments will be placed into the corresponding fields of the event counter structure it is initializing. The *group* and *name* arguments must be constant strings.

The following is a brief description of each function in the framework:

void evcnt_attach_dynamic(*struct evcnt *ev*, *int type*, *const struct evcnt *parent*, *const char *group*, *const char *name*)

Attach the event counter structure pointed to by *ev* to the system event list. The event counter is cleared and its fields initialized using the arguments to the function call. The contents of the remaining elements in the structure (e.g., the name lengths) are calculated, and the counter is added to the system event list.

The strings specified as the group and counter names must persist (with the same value) throughout the life of the event counter; they are referenced by, not copied into, the counter.

void evcnt_attach_static(*struct evcnt *ev*)

Attach the statically-initialized event counter structure pointed to by *ev* to the system event list. The event counter is assumed to be statically initialized using the **EVCNT_INITIALIZER**() macro. This function simply calculates structure elements' values as appropriate (e.g., the string lengths), and adds the counter to the system event list.

void evcnt_detach(*struct evcnt *ev*)

Detach the event counter structure pointed to by *ev* from the system event list.

Note that no method is provided to increment the value of an event counter. Code incrementing an event counter should do so by directly accessing its *ev_count* field in a manner that is known to be safe. For instance, additions to a device's event counters in the interrupt handler for that device will often be safe without additional protection (because interrupt handler entries for a given device have to be serialized). However, for other uses of event counters, additional locking or use of machine-dependent atomic operation may be appropriate. (The overhead of using a mechanism that is guaranteed to be safe to increment every counter, regardless of actual need for such a mechanism where the counter is being incremented, would be too great. On some systems, it might involve a global lock and several function calls.)

USING THE FRAMEWORK

This section includes a description on basic use of the framework and example usage of its functions.

Device drivers can use the **evcnt_attach_dynamic**() and **evcnt_detach**() functions to manage device-specific event counters. Statically configured system modules can use **evcnt_attach_static**() to configure global event counters. Similarly, loadable modules can use **evcnt_attach_static**() to configure their global event counters, **evcnt_attach_dynamic**() to attach device-specific event counters, and **evcnt_detach**() to detach all counters when being unloaded.

Device drivers that wish to use the generic event counter framework should place event counter structures in their "softc" structures. For example, to keep track of the number of interrupts for a given device (broken down further into "device readable" and "device writable" interrupts) a device driver might use:

```
struct foo_softc {
        struct device sc_dev;         /* generic device information */
        [ . . . ]
        struct evcnt sc_ev_intr;      /* interrupt count */
        struct evcnt sc_ev_intr_rd;   /* 'readable' interrupt count */
```

```
        struct evcnt sc_ev_intr_wr;   /* 'writable' interrupt count */
        [ . . . ]
};
```

In the device attach function, those counters would be registered with the system using the **evcnt_attach_dynamic**() function, using code like:

```
void
fooattach(parent, self, aux)
        struct device *parent, *self;
        void *aux;
{
        struct foo_softc *sc = (struct foo_softc *)self;

        [ . . . ]

        /* Initialize and attach event counters. */
        evcnt_attach_dynamic(&sc->sc_ev, EVCNT_TYPE_INTR,
            NULL, sc->sc_dev.dv_xname, "intr");
        evcnt_attach_dynamic(&sc->sc_ev_rd, EVCNT_TYPE_INTR,
            &sc->sc_ev, sc->sc_dev.dv_xname, "intr rd");
        evcnt_attach_dynamic(&sc->sc_ev_wr, EVCNT_TYPE_INTR,
            &sc->sc_ev, sc->sc_dev.dv_xname, "intr wr");

        [ . . . ]
}
```

If the device can be detached from the system, its detach function should invoke **evcnt_detach**() on each attached counter (making sure to detach any "parent" counters only after detaching all children).

Code like the following might be used to initialize a static event counter (in this example, one used to track CPU alignment traps):

```
        struct evcnt aligntrap_ev = EVCNT_INITIALIZER(EVCNT_TYPE_MISC,
            NULL, "cpu", "aligntrap")
```

To attach this event counter, code like the following could be used:

```
        evcnt_attach_static(&aligntrap_ev);
```

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the event counter framework can be found. All pathnames are relative to `/usr/src`.

The event counter framework itself is implemented within the file `sys/kern/subr_evcnt.c`. Data structures and function prototypes for the framework are located in `sys/sys/device.h`.

Event counters are used throughout the system.

The `vmstat`(1) source file `usr.bin/vmstat/vmstat.c` shows an example of how to access event counters from user programs.

SEE ALSO

`vmstat`(1)

HISTORY

A set of interrupt counter interfaces with similar names to the interfaces in the NetBSD generic event counter framework appeared as part of the new autoconfiguration system in 4.4BSD. Those interfaces were never widely adopted in NetBSD because of limitations in their applicability. (Their use was limited to non-hierarchical, dynamically attached device interrupt counters.) The NetBSD generic event counter framework first appeared in NetBSD 1.5.

AUTHORS

The NetBSD generic event counter framework was designed and implemented by Chris Demetriou ⟨cgd@NetBSD.org⟩.

NAME

extattr — file system extended attributes

SYNOPSIS

```
#include <sys/param.h>
#include <sys/vnode.h>
#include <sys/extattr.h>
```

DESCRIPTION

Extended attributes allow additional meta-data to be associated with vnodes representing files and directories. The semantics of this additional data is that of a "name=value" pair, where a name may be defined or undefined, and if defined, associated with zero or more bytes of arbitrary binary data. Extended attribute names exist within a set of namespaces; each operation on an extended attribute is required to provide the namespace to which the operation refers. If the same name is present in multiple namespaces, the extended attributes associated with the names are stored and manipulated independently. The following two namespaces are defined universally, although individual file systems may implement additional namespaces, or not implement these namespaces: `EXTATTR_NAMESPACE_USER`, `EXTATTR_NAMESPACE_SYSTEM`. The semantics of these attributes are intended to be as follows: user attribute data is protected according the normal discretionary and mandatory protections associated with the data in the file or directory; system attribute data is protected such that appropriate privilege is required to directly access or manipulate these attributes.

Reads of extended attribute data may return specific contiguous regions of the meta-data, in the style of `VOP_READ`(9), but writes will replace the entire current "value" associated with a given name. As there are a plethora of file systems with differing extended attributes, availability and functionality of these functions may be limited, and they should be used with awareness of the underlying semantics of the supporting file system. Authorization schemes for extended attribute data may also vary by file system, as well as maximum attribute size, and whether or not any or specific new attributes may be defined.

Extended attributes are named using a nul-terminated character string. Depending on underlying file system semantics, this name may or may not be case-sensitive. Appropriate vnode extended attribute calls are: `VOP_GETEXTATTR`(9), `VOP_LISTEXTATTR`(9), and `VOP_SETEXTATTR`(9).

SEE ALSO

`vfsops`(9), `vnodeops`(9)

NAME

extent, **extent_create**, **extent_destroy**, **extent_alloc**, **extent_alloc_subregion**, **extent_alloc_region**, **extent_free**, **extent_print** — general purpose extent manager

SYNOPSIS

```
#include <sys/malloc.h>
#include <sys/extent.h>

struct extent *
extent_create(char *name, u_long start, u_long end, int mtype,
    void *storage, size_t storagesize, int flags);

void
extent_destroy(struct extent *ex);

int
extent_alloc(struct extent *ex, u_long size, u_long alignment,
    u_long boundary, int flags, u_long *result);

int
extent_alloc_subregion(struct extent *ex, u_long substart, u_long subend,
    u_long size, u_long alignment, u_long boundary, u_long flags,
    u_long *result);

int
extent_alloc1(struct extent *ex, u_long size, u_long alignment,
    u_long skew, u_long boundary, int flags, u_long *result);

int
extent_alloc_subregion1(struct extent *ex,u_long substart,u_long subend,
    u_long size,u_long alignment,u_long skew,u_long boundary,
    u_long flags,u_long *result);

int
extent_alloc_region(struct extent *ex, u_long start, u_long size,
    int flags);

int
extent_free(struct extent *ex, u_long start, u_long size, int flags);

void
extent_print(struct extent *ex);
```

DESCRIPTION

The NetBSD extent manager provides management of areas of memory or other number spaces (such as I/O ports). An opaque structure called an **extent map** keeps track of allocated regions within the number space.

extent_create() creates an extent map managing the space from *start* to *end* inclusive. All memory allocation will use the memory type *mtype* (see malloc(9)). The extent map will have the name *name*, used for identification in case of an error. If the flag EX_NOCOALESCE is specified, only entire regions may be freed within the extent map, but internal coalescing of regions is disabled so that **extent_free**() will never have to allocate a region descriptor and therefore will never fail. The caller must specify one of the flags EX_NOWAIT or EX_WAITOK, specifying whether it is okay to wait for memory allocated for extent map overhead.

There are some applications which may want to use an extent map but can't use **malloc**() and **free**(). These applications may provide pre-allocated storage for all descriptor overhead with the arguments *storage* and *storagesize*. An extent of this type is called a **fixed extent**. If the application can safely use **malloc**() and **free**(), *storage* should be NULL. A fixed extent has a fixed number of region descriptors, so care should be taken to provide enough storage for them; alternatively, the flag EX_MALLOCOK may be passed to allocation requests to indicate that a fixed extent map may be extended using a call to **malloc**().

extent_destroy() destroys the extent map *ex*, freeing all allocated regions. If the extent is not a fixed extent, the region and internal extent descriptors themselves are freed. This function always succeeds.

extent_alloc() allocates a region in extent *ex* of size *size* that fits the provided parameters. There are two distinct allocation policies, which are selected by the *flags* argument:

EX_FAST Allocate the first region that fits the provided parameters, regardless of resulting extent fragmentation.

default Allocate the smallest region that is capable of holding the request, thus minimizing fragmentation of the extent.

The caller must specify if waiting for space in the extent is allowed using the flag EX_WAITSPACE. If EX_WAITSPACE is not specified, the allocation will fail if the request can not be satisfied without sleeping. The caller must also specify, using the EX_NOWAIT or EX_WAITOK flags, if waiting for overhead allocation is allowed. The request will be aligned to *alignment* boundaries. Alignment values must be a power of 2. If no alignment is necessary, the value 1 should be specified. If *boundary* is nonzero, the allocated region will not cross any of the numbers which are a multiple of *boundary*. If the caller specifies the EX_BOUNDZERO flag, the boundary lines begin at zero. Otherwise, the boundary lines begin at the beginning of the extent. The allocated region may begin on a boundary address, but the end of the region will not touch nor cross it. A boundary argument smaller than the size of the request is invalid. Upon successful completion, $*$*result* will contain the start of the allocated region.

extent_alloc_subregion() is similar to **extent_alloc**(), but it allows the caller to specify that the allocated region must fall within the subregion from *substart* to *subend* inclusive. The other arguments and the return values of **extent_alloc_subregion**() are otherwise the same as those of **extent_alloc**().

extent_alloc_region() allocates the specific region in the extent map *ex* beginning at *start* with the size *size*. The caller must specify whether it is okay to wait for the indicated region to be free using the flag EX_WAITSPACE. If EX_WAITSPACE is not specified, the allocation will fail if the request can not be satisfied without sleeping. The caller must also specify, using the EX_NOWAIT or EX_WAITOK flags, if waiting for overhead allocation is allowed.

The **extent_alloc1**() and **extent_alloc_subregion1**() functions are extensions that take one additional argument, *skew*, that modifies the requested alignment result in the following way: the value (*result - skew*) is aligned to *alignment* boundaries. *skew* must be a smaller number than *alignment*. Also, a boundary argument smaller than the sum of the requested skew and the size of the request is invalid.

extent_free() frees a region of *size* bytes in extent *ex* starting at *start*. If the extent has the EX_NOCOALESCE property, only entire regions may be freed. If the extent has the EX_NOCOALESCE property and the caller attempts to free a partial region, behavior is undefined. The caller must specify one of the flags EX_NOWAIT or EX_WAITOK to specify whether waiting for memory is okay; these flags have meaning in the event that allocation of a region descriptor is required during the freeing process. This situation occurs only when a partial region that begins and ends in the middle of another region is freed. Behavior is undefined if invalid arguments are provided.

extent_print() Print out information about extent *ex*. This function always succeeds. Behavior is undefined if invalid arguments are provided.

LOCKING

The extent manager performs all necessary locking on the extent map itself, and any other data structures internal to the extent manager. The locks used by the extent manager are simplelocks, and will never sleep (see lock(9)). This should be taken into account when designing the locking protocol for users of the extent manager.

RETURN VALUES

The behavior of all extent manager functions is undefined if given invalid arguments. **extent_create**() returns the extent map on success, or NULL if it fails to allocate storage for the extent map. It always succeeds when creating a fixed extent or when given the flag EX_WAITOK. **extent_alloc**(), **extent_alloc_region**(), **extent_alloc_subregion**(), and **extent_free**() return one of the following values:

0	Operation was successful.
ENOMEM	If EX_NOWAIT is specified, the extent manager was not able to allocate a region descriptor for the new region or to split a region when freeing a partial region.
EAGAIN	Requested region is not available and EX_WAITSPACE was not specified.
EINTR	Process received a signal while waiting for the requested region to become available in the extent. Does not apply to **extent_free**().

EXAMPLES

Here is an example of a (useless) function that uses several of the extent manager routines.

```
void
func()
{
        struct extent *foo_ex;
        u_long region_start;
        int error;

        /*
         * Extent "foo" manages a 256k region starting at 0x0 and
         * only allows complete regions to be freed so that
         * extent_free() never needs to allocate memory.
         */
        foo_ex = extent_create("foo", 0x0, 0x3ffff, M_DEVBUF,
            NULL, 0, EX_WAITOK | EX_NOCOALESCE);

        /*
         * Allocate an 8k region, aligned to a 4k boundary, which
         * does not cross any of the 3 64k boundaries (at 64k,
         * 128k, and 192k) within the extent.
         */
        error = extent_alloc(foo_ex, 0x2000, 0x1000, 0x10000,
            EX_NOWAIT, &region_start);
        if (error)
                panic("you lose");
```

```
        /*
         * Give up the extent.
         */
        extent_destroy(foo_ex);
}
```

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the extent manager can be found. All pathnames are relative to `/usr/src`.

The extent manager itself is implemented within the file `sys/kern/subr_extent.c`. Function prototypes for the framework are located in `sys/sys/extent.h`.

The i386 bus management code uses the extent manager for managing I/O ports and I/O memory. This code is in the file `sys/arch/i386/i386/machdep.c`.

SEE ALSO

`malloc`(9)

HISTORY

The NetBSD extent manager appeared in NetBSD 1.3.

AUTHORS

The NetBSD extent manager was architected and implemented by Jason R. Thorpe ⟨thorpej@NetBSD.org⟩.
Matthias Drochner ⟨drochner@zelux6.zel.kfa-juelich.de⟩ contributed to the initial testing and optimization of the implementation.
Chris Demetriou ⟨cgd@NetBSD.org⟩ contributed many architectural suggestions.

NAME

fetch, **fubyte**, **fuibyte**, **fusword**, **fuswintr**, **fuword**, **fuiword** — fetch data from user-space

SYNOPSIS

```
#include <sys/types.h>
#include <sys/systm.h>

int
fubyte(const void *base);

int
fusword(const void *base);

int
fuswintr(const void *base);

long
fuword(const void *base);
```

DESCRIPTION

The **fetch** functions are designed to copy small amounts of data from user-space.

The **fetch** routines provide the following functionality:

fubyte() Fetches a byte of data from the user-space address *base*.

fusword() Fetches a short word of data from the user-space address *base*.

fuswintr() Fetches a short word of data from the user-space address *base*. This function is safe to call during an interrupt context.

fuword() Fetches a word of data from the user-space address *base*.

RETURN VALUES

The **fetch** functions return the data fetched or -1 on failure. Note that these functions all do "unsigned" access, and therefore will never sign extend byte or short values. This prevents ambiguity with the error return value for all functions except **fuword**().

SEE ALSO

copy(9), store(9)

BUGS

The function **fuword**() has no way to unambiguously signal an error, because the data it reads might legitimately be the same as the -1 used to indicate an error. The other functions do not have this problem because the unsigned values returned by those can never match the -1 error return value.

NAME

file, **closef**, **ffree**, **FILE_IS_USABLE**, **FILE_USE**, **FILE_UNUSE**, **FILE_SET_MATURE** — operations on file entries

SYNOPSIS

```
#include <sys/file.h>

int
closef(struct file *fp, struct lwp *l);

void
ffree(struct file *fp);

int
FILE_IS_USABLE(struct file *fp);

void
FILE_USE(struct file *fp);

void
FILE_UNUSE(struct file *fp, struct lwp *l);

void
FILE_SET_MATURE(struct file *fp);
```

DESCRIPTION

The file descriptor table of a process references a file entry for each file used by the kernel. See `filedesc`(9) for details of the file descriptor table. Each file entry is given by:

```
struct file {
        LIST_ENTRY(file) f_list;        /* list of active files */
        int             f_flag;
        int             f_iflags;       /* internal flags */
        int             f_type;         /* descriptor type */
        u_int           f_count;        /* reference count */
        u_int           f_msgcount;     /* message queue references */
        int             f_usecount;     /* number active users */
        kauth_cred_t    f_cred;         /* creds associated with descriptor */
        struct fileops {
                int (*fo_read)(struct file *fp, off_t *offset,
                        struct uio *uio, kauth_cred_t cred, int flags);
                int (*fo_write)(struct file *fp, off_t *offset,
                        struct uio *uio, kauth_cred_t cred, int flags);
                int (*fo_ioctl)(struct file *fp, u_long com, void *data,
                        struct lwp *l);
                int (*fo_fcntl)(struct file *fp, u_int com, void *data,
                        struct lwp *l);
                int (*fo_poll)(struct file *fp, int events,
                        struct lwp *l);
                int (*fo_stat)(struct file *fp, struct stat *sp,
                        struct lwp *l);
                int (*fo_close)(struct file *fp, struct lwp *l);
        } *f_ops;
        off_t           f_offset;
        void           *f_data;         /* descriptor data */
```

```
};
```

NetBSD treats file entries in an object-oriented fashion after they are created. Each entry specifies the object type, *f_type*, which can have the values `DTYPE_VNODE`, `DTYPE_SOCKET`, `DTYPE_PIPE` and `DTYPE_MISC`. The file entry also has a pointer to a data structure, *f_data*, that contains information specific to the instance of the underlying object. The data structure is opaque to the routines that manipulate the file entry. Each entry also contains an array of function pointers, *f_ops*, that translate the generic operations on a file descriptor into the specific action associated with its type. A reference to the data structure is passed as the first parameter to a function that implements a file operation. The operations that must be implemented for each descriptor type are read, write, ioctl, fcntl, poll, stat, and close. See `vnfileops`(9) for an overview of the vnode file operations. All state associated with an instance of an object must be stored in that instance's data structure; the underlying objects are not permitted to manipulate the file entry themselves.

For data files, the file entry points to a `vnode`(9) structure. Pipes and sockets do not have data blocks allocated on the disk and are handled by the special-device filesystem that calls appropriate drivers to handle I/O for them. For pipes, the file entry points to a system block that is used during data transfer. For sockets, the file entry points to a system block that is used in doing interprocess communications.

The descriptor table of a process (and thus access to the objects to which the descriptors refer) is inherited from its parent, so several different processes may reference the same file entry. Thus, each file entry has a reference count, *f_count*. Each time a new reference is created, the reference count is incremented. When a descriptor is closed, the reference count is decremented. When the reference count drops to zero, the file entry is freed.

Some file descriptor semantics can be altered through the *`flags`* argument to the `open`(2) system call. These flags are recorded in *f_flags* member of the file entry. For example, the flags record whether the descriptor is open for reading, writing, or both reading and writing. The following flags and their corresponding `open`(2) flags are:

FAPPEND	`O_APPEND`
FASYNC	`O_ASYNC`
O_FSYNC	`O_SYNC`
FNDELAY	`O_NONBLOCK`
O_NDELAY	`O_NONBLOCK`
FNONBLOCK	`O_NONBLOCK`
FFSYNC	`O_SYNC`
FDSYNC	`O_DSYNC`
FRSYNC	`O_RSYNC`
FALTIO	`O_ALT_IO`

Some additional state-specific flags are recorded in the *f_iflags* member. Valid values include:

FIF_WANTCLOSE
: If set, then the reference count on the file is zero, but there were multiple users of the file. This can happen if a file descriptor table is shared by multiple processes. This flag notifies potential users that the file is closing and will prevent them from adding additional uses to the file.

FIF_LARVAL
: The file entry is not fully constructed (mature) and should not be used.

The `read`(2) and `write`(2) system calls do not take an offset in the file as an argument. Instead, each read or write updates the current file offset, *f_offset* in the file according to the number of bytes transferred. Since more than one process may open the same file and each needs its own offset in the file, the offset cannot be stored in the per-object data structure.

FUNCTIONS

closef(*fp*, *l*)

The internal form of close(2) which decrements the reference count on file entry *fp*. The **closef**() function release all locks on the file owned by lwp *l*, decrements the reference count on the file entry, and invokes **ffree**() to free the file entry.

ffree(*struct file *fp*)

Free file entry *fp*. The file entry was created in falloc(9).

FILE_IS_USABLE(*fp*)

Ensure that the file entry is useable by ensuring that neither the FIF_WANTCLOSE and FIF_LARVAL flags are not set in *f_iflags*.

FILE_USE(*fp*)

Increment the reference count on file entry *fp*.

FILE_UNUSE(*fp*, *l*)

Decrement the reference count on file entry *fp*. If the FIF_WANTCLOSE flag is set in *f_iflags*, the file entry is freed.

FILE_SET_MATURE(*fp*)

Mark the file entry as being fully constructed (mature) by clearing the FIF_LARVAL flag in *f_iflags*.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using file entries can be found. All pathnames are relative to /usr/src.

The framework for file entry handling is implemented within the file sys/kern/kern_descrip.c.

SEE ALSO

dofileread(9), filedesc(9), vnfileops(9), vnode(9)

NAME

fileassoc — in-kernel, file-system independent, file-meta data association

SYNOPSIS

```
#include <sys/fileassoc.h>
```

DESCRIPTION

The **fileassoc** KPI allows association of meta-data with files independent of file-system support for such elaborate meta-data.

When plugging a new fileassoc to the system, a developer can specify private data to be associated with every file, as well as (potentially different) private data to be associated with every file-system mount.

For example, a developer might choose to associate a custom ACL with every file, and a count of total files with ACLs with the mount.

Kernel Programming Interface

Designed with simplicity in mind, the **fileassoc** KPI usually accepts four different types of parameters to the most commonly used routines:

*struct mount * mp*
: Describing a mount on which to take action.

*struct vnode * vp*
: Describing a file on which to take action.

fileassoc_t id
: Describing an id, as returned from a successful call to **fileassoc_register**().

*void * data*
: Describing a custom private data block, attached to either a file or a mount.

Before using the **fileassoc** KPI it is important to keep in mind that the interface provides memory management only for **fileassoc** internal memory. Any additional memory stored in the tables (such as private data-structures used by custom fileassocs) should be allocated and freed by the developer.

fileassoc provides the ability to specify a "cleanup" routine to **fileassoc_register**() (see below) to be called whenever an entry for a file or a mount is deleted.

Fileassoc Registration and Deregistration Routines

These routines allow a developer to allocate a **fileassoc** slot to be used for private data.

int **fileassoc_register**(*const char *name*, *fileassoc_cleanup_cb_t cleanup_cb*, *fileassoc_t *result*)
: Registers a new fileassoc as *name*, and returns a *fileassoc_t* via *result* to be used as identifier in subsequent calls to the **fileassoc** subsystem.

  fileassoc_register() returns zero on success. Otherwise, an error number will be returned.

  If *cleanup_cb* is not NULL, it will be called during delete/clear operations (see routines below) with indication whether the passed data is file- or mount-specific.

  cleanup_cb should be a function receiving a *void * * and returning *void*. See the **EXAMPLES** section for illustration.

int **fileassoc_deregister**(*fileassoc_t id*)
: Deregisters a **fileassoc** whose id is *id*.

Note that calling **fileassoc_deregister**() only frees the associated slot in the **fileassoc** subsystem. It is up to the developer to take care of garbage collection.

Lookup Routines

These routines allow lookup of **fileassoc** mounts, files, and private data attached to them.

void * **fileassoc_lookup**(*struct vnode *vp*, *fileassoc_t id*)
Returns the private data for the file/id combination or NULL if not found.

Mount-wide Routines

int **fileassoc_table_delete**(*struct mount *mp*)
Deletes a fileassoc table for *mp*.

int **fileassoc_table_clear**(*struct mount *mp*, *fileassoc_t id*)
Clear all table entries for *fileassoc* from *mp*.

If specified, the fileassoc's "cleanup routine" will be called with a pointer to the private data-structure.

int **fileassoc_table_run**(*struct mount *mp*, *fileassoc_t id*, *fileassoc_cb_t cb*, *void *cookie*)
For each entry for *id*, call *cb* with the entry being the first argument, and *cookie* being the second argument.

cb is a function returning *void* and receiving one *void* * parameter.

File-specific Routines

int **fileassoc_file_delete**(*struct vnode *vp*)
Delete the fileassoc entries for *vp*.

If specified, the "cleanup routines" of all fileassoc types added will be called with a pointer to the corresponding private data structure and indication of FILEASSOC_CLEANUP_FILE.

Fileassoc-specific Routines

int **fileassoc_add**(*struct vnode *vp*, *fileassoc_t id*, *void *data*)
Add private data in *data* for *vp*, for the fileassoc specified by *id*.

If a table for the mount-point *vp* is on doesn't exist, one will be created automatically. **fileassoc** manages internally the optimal table sizes as tables are modified.

int **fileassoc_clear**(*struct vnode *vp*, *fileassoc_t id*)
Clear the private data for *vp*, for the fileassoc specified by *id*.

If specified, the fileassoc's "cleanup routine" will be called with a pointer to the private data-structure and indication of FILEASSOC_CLEANUP_FILE.

EXAMPLES

The following code examples should give you a clue on using **fileassoc** for your purposes.

First, we'll begin with registering a new id. We need to do that to save a slot for private data storage with each mount and/or file:

```
fileassoc_t myhook_id;
int error;

error = fileassoc_register("my_hook", myhook_cleanup, &myhook_id);
if (error != 0)
```

```
        ...handle error...
```

In the above example we pass a **myhook_cleanup**() routine. It could look something like this:

```
void
myhook_cleanup(void *data)
{

        printf("Myhook: Removing entry for file.0);
        ...handle file entry removal...
        free(data, M_TEMP);
}
```

Another useful thing would be to add our private data to a file. For example, let's assume we keep a custom ACL with each file:

```
int
myhook_acl_add(struct vnode *vp, struct myhook_acl *acl)
{
        int error;

        error = fileassoc_add(vp, myhook_id, acl);
        if (error) {
                printf("Myhook: Could not add ACL.0);
                ...handle error...
        }

        printf("Myhook: Added ACL.0);

        return (0);
}
```

Adding an entry will override any entry that previously exists.

Whatever your plug is, eventually you'll want to access the private data you store with each file. To do that you can use the following:

```
int
myhook_acl_access(struct vnode *vp, int access_flags)
{
        struct myhook_acl *acl;

        acl = fileassoc_lookup(vp, myhook_id);
        if (acl == NULL)
                return (0);

        error = myhook_acl_eval(acl, access_flags);
        if (error) {
                printf("Myhook: Denying access based on ACL decision.0);
                return (error);
        }

        return (0);
}
```

And, in some cases, it may be desired to remove private data associated with an file:

```
int error;

error = fileassoc_clear(vp, myhook_id);
if (error) {
        printf("Myhook: Error occurred during fileassoc removal.0);
        ...handle error...
}
```

As mentioned previously, the call to **fileassoc_clear**() will result in a call to the "cleanup routine" specified in the initial call to **fileassoc_register**().

The above should be enough to get you started.

For example usage of **fileassoc**, see the Veriexec code.

CODE REFERENCES

`src/sys/kern/kern_fileassoc.c`

HISTORY

The **fileassoc** KPI first appeared in NetBSD 4.0.

AUTHORS

Elad Efrat ⟨elad@NetBSD.org⟩
Brett Lymn ⟨blymn@NetBSD.org⟩

NAME

filedesc, **dupfdopen**, **falloc**, **fd_getfile**, **fdalloc**, **fdcheckstd**, **fdclear**, **fdclone**, **fdcloseexec**, **fdcopy**, **fdexpand**, **fdfree**, **fdinit**, **fdrelease**, **fdremove**, **fdshare**, **fdunshare** — file descriptor tables and operations

SYNOPSIS

```
#include <sys/file.h>
#include <sys/filedesc.h>

int
falloc(struct lwp *l, struct file **resultfp, int *resultfd);

struct file *
fd_getfile(struct filedesc *fdp, int fd);

int
dupfdopen(struct lwp *l, int indx, int dfd, int mode, int error);

int
fdalloc(struct proc *p, int want, int *result);

int
fdcheckstd(struct lwp *l);

void
fdclear(struct lwp *l);

int
fdclone(struct lwp *l, struct file *fp, int fd, int flag,
    const struct fileops *fops, void *data);

void
fdcloseexec(struct lwp *l);

struct filedesc *
fdcopy(struct proc *p);

void
fdexpand(struct proc *p);

void
fdfree(struct lwp *l);

struct filedesc *
fdinit(struct proc *p);

int
fdrelease(struct lwp *l, int fd);

void
fdremove(struct filedesc *fdp, int fd);

void
fdshare(struct proc *p1, struct proc *p2);

void
fdunshare(struct lwp *l);
```

DESCRIPTION

For user processes, all I/O is done through file descriptors. These file descriptors represent underlying objects supported by the kernel and are created by system calls specific to the type of object. In NetBSD, six types of objects can be represented by file descriptors: data files, pipes, sockets, event queues, crypto, and miscellaneous.

The kernel maintains a descriptor table for each process which is used to translate the external representation of a file descriptor into an internal representation. The file descriptor is merely an index into this table. The file descriptor table maintains the following information:

- the number of descriptors allocated in the file descriptor table;
- approximate next free descriptor;
- a reference count on the file descriptor table; and
- an array of open file entries.

On creation of the file descriptor table, a fixed number of file entries are created. It is the responsibility of the file descriptor operations to expand the available number of entries if more are required. Each file entry in the descriptor table contains the information necessary to access the underlying object and to maintain common information. See `file`(9) for details of operations on the file entries.

New file descriptors are generally allocated by **`falloc`**() and freed by **`fdrelease`**(). File entries are extracted from the file descriptor table by **`fd_getfile`**(). Most of the remaining functions in the interface are purpose specific and perform lower-level file descriptor operations.

FUNCTIONS

The following functions are high-level interface routines to access the file descriptor table for a process and its file entries.

`falloc`(*p*, **resultfp*, **resultfd*)

Create a new open file entry and allocate a file descriptor for process *p*. This operation is performed by invoking **`fdalloc`**() to allocate the new file descriptor. The credential on the file entry are inherited from process *p*. The **`falloc`**() function is responsible for expanding the file descriptor table when necessary.

A pointer to the file entry is returned in **resultfp* and the file descriptor is returned in **resultfd*. The **`falloc`**() function returns zero on success, otherwise an appropriate error is returned.

`fd_getfile`(*fdp*, *fd*)

Get the file entry for file descriptor *fd* in the file descriptor table *fdp*. The file entry is returned if it is valid and useable, otherwise `NULL` is returned.

`dupfdopen`(*l*, *indx*, *dfd*, *mode*, *error*)

Duplicate file descriptor *dfd* for lwp *l*.

The following functions operate on the file descriptor table for a process.

`fdalloc`(*p*, *want*, **result*)

Allocate a file descriptor *want* for process *p*. The resultant file descriptor is returned in **result*. The **`fdalloc`**() function returns zero on success, otherwise an appropriate error is returned.

`fdcheckstd`(*l*)

Check the standard file descriptors 0, 1, and 2 and ensure they are referencing valid file descriptors. If they are not, create references to `/dev/null`. This operation is necessary as these file descriptors are given implicit significance in the Standard C Library and it is unsafe for `setuid`(2) and `setgid`(2) processes to be started with these file descriptors closed.

fdclear(*l*)
Clear the descriptor table for lwp *l*. This operation is performed by invoking **fdinit**() to initialise a new file descriptor table to replace the old file descriptor table and invoking **fdfree**() to release the old one.

fdclone(*l*, *fp*, *fd*, *flag*, *fops*, *data*)
This function is meant to be used by devices which allocate a file entry upon open. **fdclone**() fills *fp* with the given parameters. It always returns the in-kernel errno value EMOVEFD, which is meant to be returned from the device open routine. This special return value is interpreted by the caller of the device open routine.

fdcloseexec(*l*)
Close any files for process *p* that are marked "close on exec". This operation is performed by invoking **fdunshare**() for the process and invoking **fdrelease**() on the appropriate file descriptor.

fdcopy(*p*)
Copy the file descriptor table from process *p* and return a pointer to the copy. The returned file descriptor is guaranteed to have a reference count of one. All file descriptor state is maintained. The reference counts on each file entry referenced by the file descriptor table is incremented accordingly.

fdexpand(*p*)
Expand the file descriptor table for process *p* by allocating memory for additional file descriptors.

fdfree(*l*)
Decrement the reference count on the file descriptor table for lwp *l* and release the file descriptor table if the reference count drops to zero.

fdinit(*p*)
Create a file descriptor table using the same current and root directories of process *p*. The returned file descriptor table is guaranteed to have a reference count of one.

fdrelease(*l*, *fd*)
Remove file descriptor *fd* from the file descriptor table of lwp *l*. The operation is performed by invoking **closef**().

fdremove(*fdp*, *fd*)
Unconditionally remove the file descriptor *fd* from file descriptor table *fdp*.

fdshare(*p1*, *p2*)
Share the file descriptor table belonging to process *p1* with process *p2*. Process *p2* is assumed not to have a file descriptor table already allocated. The reference count on the file descriptor table is incremented. This function is used by fork1(9).

fdunshare(*l*)
Ensure that lwp *l* does not share its file descriptor table. If its file descriptor table has more than one reference, the file descriptor table is copied by invoking **fdcopy**(). The reference count on the original file descriptor table is decremented.

RETURN VALUES

Successful operations return zero. A failed operation will return a non-zero return value. Possible values include:

[EBADF] Bad file descriptor specified.

[EMFILE] Cannot exceed file descriptor limit.

[ENOSPC] No space left in file descriptor table.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using file descriptors can be found. All pathnames are relative to `/usr/src`.

The framework for file descriptor handling is implemented within the file `sys/kern/kern_descrip.c`.

SEE ALSO

`file`(9)

NAME

firmload — Firmware loader API for device drivers

SYNOPSIS

```
#include <dev/firmload.h>

int
firmware_open(const char *drvname, const char *imgname,
    firmware_handle_t *fhp);

int
firmware_close(firmware_handle_t fh);

off_t
firmware_get_size(firmware_handle_t fh);

int
firmware_read(firmware_handle_t fh, off_t offset, void *buf, size_t size);

void *
firmware_malloc(size_t size);

void
firmware_free(void *buf, size_t size);
```

DESCRIPTION

firmload provides a simple and convenient API for device drivers to load firmware images from files residing in the file system that are necessary for the devices that they control. Firmware images reside in sub-directories, one for each driver, of a series of colon-separated path prefixes specified the sysctl variable `hw.firmware.path`.

The following functions are provided by the **firmload** API:

int **firmware_open**(*const char *drvname*, *const char *imgname*, *firmware_handle_t *fhp*)

Open then firmware image *imgname* for the driver *drvname*. The path to the firmware image file is constructed by appending the string "/drvname/imgname" to each configured path prefix until opening the firmware image file succeeds. Upon success, **firmware_open**() returns 0 and stores a firmware image handle in the location pointed to by *fhp*. Otherwise, an error code is returned to indicate the reason for failure.

int **firmware_close**(*firmware_handle_t fh*)

Close the firmware image file associated with the firmware handle *fh*. Returns 0 upon success or an error code to indicate the reason for failure.

off_t **firmware_get_size**(*firmware_handle_t fh*)

Returns the size of the image file associated with the firmware handle *fh*.

int **firmware_read**(*firmware_handle_t fh*, *off_t offset*, *void *buf*, *size_t size*)

Reads from the image file associated with the firmware handle *fh* beginning at offset *offset* for length *size*. The firmware image data is placed into the buffer specified by *buf*. Returns 0 upon success or an error code to indicate the reason for failure.

void *__firmware_malloc__(*size_t size*)

Allocates a region of wired kernel memory of size *size*. Note: **firmware_malloc**() may block.

void **firmware_free**(*void *buf*, *size_t size*)

Frees a region of memory previously allocated by **firmware_malloc**().

SEE ALSO

autoconf(9), malloc(9), vnsubr(9)

HISTORY

The **firmload** framework first appeared in NetBSD 4.0.

AUTHORS

Jason Thorpe ⟨thorpej@NetBSD.org⟩

NAME

fork1 — create a new process

SYNOPSIS

```
#include <sys/types.h>
#include <sys/proc.h>

int
fork1(struct lwp *l1, int flags, int exitsig, void *stack, size_t stacksize,
    void (*func)(void *), void *arg, register_t *retval,
    struct proc **rnewprocp);
```

DESCRIPTION

fork1() creates a new process out of the process behind *l1*, which is assumed to be the current lwp. This function is used primarily to implement the fork(2) and vfork(2) system calls, but is versatile enough to be used as a backend for e.g. the __clone(2) call.

The *flags* argument controls the semantics of the fork operation, and is made up of the bitwise-OR of the following values:

FORK_PPWAIT — The parent process will sleep until the child process successfully calls execve(2) or exits (either by a call to _exit(2) or abnormally).

FORK_SHAREVM — The child process will share the parent's virtual address space. If this flag is not specified, the child will get a copy-on-write snapshot of the parent's address space.

FORK_SHARECWD — The child process will share the parent's current directory, root directory, and file creation mask.

FORK_SHAREFILES — The child process will share the parent's file descriptors.

FORK_SHARESIGS — The child process will share the parent's signal actions.

FORK_NOWAIT — The child process will at creation time be inherited by the init process.

FORK_CLEANFILES — The child process will not copy or share the parent's descriptors, but rather will start out with a clean set.

A *flags* value of 0 indicates a standard fork operation.

The *exitsig* argument controls the signal sent to the parent on child death. If normal operation desired, SIGCHLD should be supplied.

It is possible to specify the child userspace stack location and size by using the *stack* and *stacksize* arguments, respectively. Values NULL and 0, respectively, will give the child the default values for the machine architecture in question.

The arguments *func* and *arg* can be used to specify a kernel function to be called when the child process returns instead of **child_return**(). These are used for example in starting the init process and creating kernel threads.

The *retval* argument is provided for the use of system call stubs. If *retval* is not NULL, it will hold the following values after successful completion of the fork operation:

retval[0] This will contain the pid of the child process.

retval[1] In the parent process, this will contain the value 0. In the child process, this will contain 1.

User level system call stubs typically subtract 1 from *retval[1]* and bitwise-AND it with *retval[0]*, thus returning the pid to the parent process and 0 to the child.

If *rnewprocp* is not NULL, ∗*rnewprocp* will point to the newly created process upon successful completion of the fork operation.

RETURN VALUES

Upon successful completion of the fork operation, **fork1**() returns 0. Otherwise, the following error values are returned:

[EAGAIN] The limit on the total number of system processes would be exceeded.

[EAGAIN] The limit RLIMIT_NPROC on the total number of processes under execution by this user id would be exceeded.

SEE ALSO

execve(2), fork(2), vfork(2)

NAME

fsetown, **fgetown**, **fownsignal** — file descriptor owner handling functions

SYNOPSIS

```
#include <sys/file.h>

int
fsetown(struct lwp *l, pid_t *pgid, int cmd, const void *data);

int
fgetown(struct lwp *l, pid_t pgid, int cmd, void *data);

void
fownsignal(pid_t pgid, int signo, int code, int band, void *fdescdata);
```

DESCRIPTION

These functions handle file descriptor owner related ioctls and related signal delivery. Device drivers and other parts of the kernel call these functions from ioctl entry functions or I/O notification functions.

fsetown() sets the owner of file. *cmd* is an ioctl command, one of SIOCSPGRP, FIOSETOWN, and TIOCSPGRP. *data* is interpreted as a pointer to a signed integer, the integer being the ID of the owner. The *cmd* determines how exactly *data* should be interpreted. If *cmd* is TIOCSPGRP, the ID needs to be positive and is interpreted as process group ID. For SIOCSPGRP and FIOSETOWN, the passed ID is the process ID if positive, or the process group ID if negative.

fgetown() returns the current owner of the file. *cmd* is an ioctl command, one of SIOCGPGRP, FIOGETOWN, and TIOCGPGRP. *data* is interpreted as a pointer to a signed integer, and the value is set according to the passed *cmd*. For TIOCGPGRP, the returned *data* value is positive process group ID if the owner is the process group, or negative process ID if the owner is a process. For other ioctls, the returned value is the positive process ID if the owner is a process, or the negative process group ID if the owner is a process group.

fownsignal() schedules the *signo* signal to be sent to the current file descriptor owner. The signals typically used with this function are SIGIO and SIGURG. The *code* and *band* arguments are sent along with the signal as additional signal specific information if SA_SIGINFO is activated. If the information is not available from the context of the **fownsignal**() call, these should be passed as zero. *fdescdata* is used to lookup the file descriptor for signals. If it is specified, the file descriptor number is sent along with the signal as additional signal specific information. If file descriptor data pointer is not available in the context of the **fownsignal**() call, NULL should be used instead.

Note that a fcntl(2) F_SETOWN request is translated by the kernel to a FIOSETOWN ioctl, and F_GETOWN is translated to FIOGETOWN. This is done transparently by generic code, before the device- or subsystem-specific ioctl entry function is called.

SEE ALSO

fcntl(2), siginfo(2), signal(7), ioctl(9), signal(9)

HISTORY

These kernel functions appeared in NetBSD 2.0.

NAME

fstrans, **fstrans_setstate**, **fstrans_getstate**, **fstrans_start**, **fstrans_start_nowait**, **fstrans_done**, **fstrans_is_owner**, **fscow_establish**, **fscow_disestablish**, **fscow_run** — file system suspension helper subsystem

SYNOPSIS

```
#include <sys/mount.h>
#include <sys/fstrans.h>

int
fstrans_setstate(struct mount *mp, enum fstrans_state new_state);

enum fstrans_state
fstrans_getstate(struct mount *mp);

void
fstrans_start(struct mount *mp, enum fstrans_lock_type lock_type);

int
fstrans_start_nowait(struct mount *mp, enum fstrans_lock_type lock_type);

void
fstrans_done(struct mount *mp);

int
fstrans_is_owner(struct mount *mp);

int
fscow_establish(struct mount *mp, int (*func)(void *, struct buf *, bool),
    void *cookie);

int
fscow_disestablish(struct mount *mp,
    int (*func)(void *, struct buf *, bool), void *cookie);

int
fscow_run(struct buf *bp, bool data_valid);
```

DESCRIPTION

The **fstrans** subsystem is a set of operations to assist file system suspension. These operations must not be used outside of file systems.

File systems supporting this subsystem must set the flag `IMNT_HAS_TRANS` in `mnt_iflag`.

File systems are always in one of these states:

`FSTRANS_NORMAL`	normal operations.
`FSTRANS_SUSPENDING`	preparing a suspension.
`FSTRANS_SUSPENDED`	suspended.

This state is represented by *`enum fstrans_state`*.

fstrans_getstate(*mp*)
: returns the current state of the file system *mp*.

fstrans_setstate(*mp*, *new_state*)
: changes the state of the file system *mp* to *new_state*.

All file system operations use a *fstrans lock*. This lock is recursive. A thread already owning a lock will always get another lock. The lock has two variants:

`FSTRANS_SHARED` this lock will be granted if the file system is in state `FSTRANS_NORMAL`.
`FSTRANS_LAZY` this lock will be granted if the file system is in state `FSTRANS_NORMAL` or `FSTRANS_SUSPENDING`. It needs special care because operations using this variant will not block while the file system prepares suspension.

The lock variant is represented by *`enum fstrans_lock_type`*.

`fstrans_start`(*`mp`*, *`lock_type`*)
sets a lock of type *`lock_type`* on the file system *`mp`*.

`fstrans_start_nowait`(*`mp`*, *`lock_type`*)
will not wait for a state change of the file system when attempting to aquire the lock. The thread may still sleep while attempting to aquire the lock.

`fstrans_done`(*`mp`*)
releases a lock on the file system *`mp`*.

`fstrans_is_owner`(*`mp`*)
returns `true` if this thread is currently suspending the file system *`mp`*.

`fscow_establish`(*`mp`*, *`func`*, *`cookie`*)
Establish a copy-on-write callback for the file system *`mp`*. *`func`* will be called for every buffer written through this file system.

`fscow_disestablish`(*`mp`*, *`func`*, *`cookie`*)
Disestablish a copy-on-write callback registered with **`fscow_establish`**().

`fscow_run`(*`bp`*, *`data_valid`*)
Run all copy-on-write callbacks established for the file system this buffer belongs to. If *`data_valid`* is `true` the buffer data has not yet been modified.

RETURN VALUES

The functions **`fstrans_setstate`**() and **`fstrans_start_nowait`**() return zero on success and an error value on failure.

EXAMPLES

The following is an example of a file system suspend operation.

```
int
xxx_suspendctl(struct mount *mp, int cmd)
{
        int error;

        switch (cmd) {
        case SUSPEND_SUSPEND:
                error = fstrans_setstate(mp, FSTRANS_SUSPENDING);
                if (error != 0)
                        return error;

                /* Sync file system state to disk. */

                return fstrans_setstate(mp, FSTRANS_SUSPENDED);

        case SUSPEND_RESUME:
                return fstrans_setstate(mp, FSTRANS_NORMAL);

        default:
```

```
                return EINVAL;
        }
}
```

This is an example of a file system operation.

```
int
xxx_create(void *v)
{
        struct vop_create_args *ap = v;
        struct mount *mp = ap->a_dvp->v_mount;
        int error;

        if ((error = fstrans_start(mp, FSTRANS_SHARED)) != 0)
                return error;

        /* Actually create the node. */

        fstrans_done(mp);

        return 0;
}
```

SEE ALSO

`vfs_resume`(9), `vfs_suspend`(9)

CODE REFERENCES

The actual code implementing this subsystem can be found in the file `sys/kern/vfs_trans.c`.

HISTORY

The **`fstrans`** subsystem appeared in NetBSD 5.0.

AUTHORS

The **`fstrans`** subsystem was written by Jürgen Hannken-Illjes ⟨hannken@NetBSD.org⟩.

NAME

getiobuf, **putiobuf** — I/O descriptor allocation interface

SYNOPSIS

```
#include <sys/buf.h>

struct buf *
getiobuf(struct vnode *vp, bool waitok);

void
putiobuf(struct buf *bp);
```

DESCRIPTION

getiobuf() allocates a *buf* structure.

vp The vnode to which the allocated buffer will be associated. This can be NULL.

waitok If true, *getiobuf* can sleep until enough memory is available. Otherwise, it returns NULL immediately if enough memory is not available.

Note that the allocated buffer doesn't belong to buffer cache. To free it, **putiobuf**() should be used. **brelse**() should not be used on it.

putiobuf() frees *bp*, which should be a buffer allocated with **getiobuf**().

SEE ALSO

buffercache(9), intro(9)

NAME

hardclock — real-time timer

SYNOPSIS

void
hardclock(*struct clockframe* *);

DESCRIPTION

The **hardclock**() function gets called hz(9) times per second. It performs different tasks:

- Run the current process's virtual and profile time (decrease the corresponding timers, if they are activated, and generate SIGVTALRM or SIGPROF, respectively).
- Increment the time-of-day, taking care of any ntpd(8) or adjtime(2) induced changes and leap seconds, as well as any necessary compensations to keep in sync with PPS signals or external clocks, if support for this is in the kernel (see options(4)).
- Schedule softclock interrupts if any callouts should be triggered (see callout(9)).

SEE ALSO

adjtime(2), ntp_adjtime(2), signal(7), ntpd(8), callout(9), hz(9)

NAME

hash, **hash32_buf**, **hash32_str**, **hash32_strn** — kernel hash functions

SYNOPSIS

```
#include <sys/types.h>
#include <sys/hash.h>

uint32_t
hash32_buf(const void *buf, size_t len, uint32_t ihash);

uint32_t
hash32_str(const void *buf, uint32_t ihash);

uint32_t
hash32_strn(const void *buf, size_t len, uint32_t ihash);
```

DESCRIPTION

The **hash** functions returns a hash of the given buffer.

The **hash32_buf**() function returns a 32 bit hash of *buf*, which is *len* bytes long, seeded with an initial hash of *ihash* (which is usually HASH32_BUF_INIT). This function may use a different algorithm to **hash32_str**() and **hash32_strn**().

The **hash32_str**() function returns a 32 bit hash of *buf*, which is a NUL terminated ASCII string, seeded with an initial hash of *ihash* (which is usually HASH32_STR_INIT). This function must use the same algorithm as **hash32_strn**(), so that the same data returns the same hash.

The **hash32_strn**() function returns a 32 bit hash of *buf*, which is a NUL terminated ASCII string, up to a maximum of *len* bytes, seeded with an initial hash of *ihash* (which is usually HASH32_STR_INIT). This function must use the same algorithm as **hash32_str**(), so that the same data returns the same hash.

The *ihash* parameter is provided to allow for incremental hashing by allowing successive calls to use a previous hash value.

RETURN VALUES

The *hash32_** functions return a 32 bit hash of the provided buffer.

HISTORY

The kernel hashing API first appeared in NetBSD 1.6.

NAME

hashinit, **hashdone** — kernel hash table construction and destruction

SYNOPSIS

```
#include <sys/systm.h>

void *
hashinit(u_int chains, enum hashtype htype, bool waitok, u_long *hashmask);

void
hashdone(void *hashtbl, enum hashtype htype, u_long hashmask);
```

DESCRIPTION

The **hashinit**() function allocates and initializes space for a simple chaining hash table. The number of slots will be the least power of two not smaller than *chains*. The customary choice for *chains* is the maximum number of elements you intend to store divided by your intended load factor. The `LIST...` or `TAILQ...` macros of queue(3) can be used to manipulate the chains; pass `HASH_LIST` or `HASH_TAILQ` as *htype* to indicate which. Each slot will be initialized as the head of an empty chain of the proper type. Because different data structures from queue(3) can define head structures of different sizes, the total size of the allocated table can vary with the choice of *htype*.

If *waitok* is true, *hashinit* can wait until enough memory is available. Otherwise, it immediately fails if there is not enough memory is available.

A value will be stored into **hashmask* suitable for masking any computed hash, to obtain the index of a chain head in the allocated table.

The **hashdone**() function deallocates the storage allocated by **hashinit**() and pointed to by *hashtbl*, given the same *htype* and *hashmask* that were passed to and returned from **hashinit**(). If the table contains any nonempty chain when **hashdone**() is called, the result is undefined.

RETURN VALUES

The value returned by **hashinit**() should be cast as pointer to an array of `LIST_HEAD` or `TAILQ_HEAD` as appropriate. **hashinit**() returns `NULL` on failure.

SEE ALSO

queue(3), hash(9), malloc(9)

CODE REFERENCES

These functions are implemented in `sys/kern/kern_hash.c`.

HISTORY

A **hashinit**() function was present, without the *htype* or *mflags* arguments, in 4.4BSD alpha. It was independent of queue(3) and simply allocated and nulled a table of pointer-sized slots. It sized the table to the *largest* power of two *not greater than* *chains*; that is, it built in a load factor between 1 and 2.

NetBSD 1.0 was the first NetBSD release to have a **hashinit**() function. It resembled that from 4.4BSD but made each slot a `LIST_HEAD` from queue(3). For NetBSD 1.3.3 it had been changed to size the table to the least power of two not less than *or equal to* *chains*. By NetBSD 1.4 it had the *mflags* argument and the current sizing rule.

NetBSD 1.5 had the **hashdone**() function. By NetBSD 1.6 **hashinit**() supported `LIST` or `TAILQ` chains selected with *htype*.

FreeBSD has a **hashinit**() with behavior equivalent (as of FreeBSD 6.1) to that in NetBSD 1.0, and a **hashdestroy**() that behaves as **hashdone**() but checks that all chains are empty first. OpenBSD has a **hashinit**() comparable (as of OpenBSD 3.9) to that of NetBSD 1.4. This manual page was added for NetBSD 4.0.

BUGS

The only part of the work of implementing a hash table that these functions relieve is the part that isn't much work.

NAME

humanize_number, **format_bytes** — format a number into a human readable form

SYNOPSIS

int
humanize_number(*char *buf*, *size_t len*, *uint64_t number*, *const char *suffix*, *int divisor*);

int
format_bytes(*char *buf*, *size_t len*, *uint64_t number*);

DESCRIPTION

humanize_number

The **humanize_number**() function formats the unsigned 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 *divisor* until it will. In this case, prefix *suffix* with the appropriate SI designator. Suitable values of *divisor* are 1024 or 1000 to remain consistent with the common meanings of the SI designator prefixes.

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*.

format_bytes

The **format_bytes**() function is a front-end to **humanize_number**() that calls the latter with a *suffix* of "B". Also, if the suffix in the returned *buffer* would not have a prefix, remove the suffix. This means that a result of "100000" occurs, instead of "100000 B".

RETURN VALUES

humanize_number() and **format_bytes**() return the number of characters stored in *buffer* (excluding the terminating NUL) upon success, or −1 upon failure.

HISTORY

humanize_number() and **format_bytes**() first appeared in NetBSD 1.5.

NAME

hz — system clock frequency

SYNOPSIS

```
#include <sys/kernel.h>
```

int hz;

DESCRIPTION

hz specifies the number of times the `hardclock`(9) timer ticks per second. **hz** is hardware-dependent; it can be overridden (if the machine dependent code supports this) by defining *HZ* in the kernel configuration file (see `options`(4)). Only override the default value if you really know what you are doing.

SEE ALSO

`options`(4), `callout`(9), `hardclock`(9), `microtime`(9), `time_second`(9)

NAME

ieee80211_ifattach, **ieee80211_ifdetach**, **ieee80211_mhz2ieee**, **ieee80211_chan2ieee**, **ieee80211_ieee2mhz**, **ieee80211_media_init**, **ieee80211_media_change**, **ieee80211_media_status**, **ieee80211_watchdog**, **ieee80211_setmode**, **ieee80211_chan2mode**, **ieee80211_rate2media**, **ieee80211_media2rate** — core 802.11 network stack functions

SYNOPSIS

```
#include <net80211/ieee80211_var.h>
#include <net80211/ieee80211_proto.h>

void
ieee80211_ifattach(struct ieee80211com *ic);

void
ieee80211_ifdetach(struct ieee80211com *ic);

u_int
ieee80211_mhz2ieee(u_int freq, u_int flags);

u_int
ieee80211_chan2ieee(struct ieee80211com *ic, struct ieee80211_channel *c);

u_int
ieee80211_ieee2mhz(u_int chan, u_int flags);

void
ieee80211_media_init(struct ieee80211com *ic,
    ifm_change_cb_t media_change,ifm_stat_cb_t media_stat);

int
ieee80211_media_change(struct ifnet *ifp);

void
ieee80211_media_status(struct ifnet *ifp, struct ifmediareq *imr);

void
ieee80211_watchdog(struct ieee80211com *ic);

int
ieee80211_setmode(struct ieee80211com *ic, enum ieee80211_phymode mode);

enum ieee80211_phymode
ieee80211_chan2mode(struct ieee80211com *ic,
    struct ieee80211_channel *chan);

int
ieee80211_rate2media(struct ieee80211com *ic,int rate,
    enum ieee80211_phymode mode);

int
ieee80211_media2rate(int mword);
```

DESCRIPTION

The **ieee80211** collection of functions are used to manage wireless network interfaces in the system which use the system's software 802.11 network stack. Most of these functions require that attachment to the stack is performed before calling. Several utility functions are also provided; these are safe to call from any driver without prior initialization.

The **ieee80211_ifattach**() function attaches the wireless network interface *ic* to the 802.11 network stack layer. This function must be called before using any of the **ieee80211** functions which need to store driver state across invocations. This function also performs Ethernet and BPF attachment (by calling **ether_ifattach**() and **bpfattach2**()) on behalf of the caller.

The **ieee80211_ifdetach**() function frees any **ieee80211** structures associated with the driver, and performs Ethernet and BPF detachment on behalf of the caller.

The **ieee80211_mhz2ieee**() utility function converts the frequency *freq* (specified in MHz) to an IEEE 802.11 channel number. The *flags* argument is a hint which specifies whether the frequency is in the 2GHz ISM band (*IEEE80211_CHAN_2GHZ*) or the 5GHz band (*IEEE80211_CHAN_5GHZ*); appropriate clipping of the result is then performed.

The **ieee80211_chan2ieee**() function converts the channel specified in **c* to an IEEE channel number for the driver *ic*. If the conversion would be invalid, an error message is printed to the system console. This function REQUIRES that the driver is hooked up to the **ieee80211** subsystem.

The **ieee80211_ieee2mhz**() utility function converts the IEEE channel number *chan* to a frequency (in MHz). The *flags* argument is a hint which specifies whether the frequency is in the 2GHz ISM band (*IEEE80211_CHAN_2GHZ*) or the 5GHz band (*IEEE80211_CHAN_5GHZ*); appropriate clipping of the result is then performed.

The **ieee80211_media_init**() function initializes media data structures used by the *ifmedia* interface for the driver *ic*. It must be called by the driver after calling **ieee80211_ifattach**() and before calling most **ieee80211** functions. The *media_change* and *media_stat* arguments specify helper functions which will be invoked by the *ifmedia* framework when the user changes or queries media options, using a command such as `ifconfig`(8).

The **ieee80211_media_status**() and **ieee80211_media_change**() functions are device-independent handlers for *ifmedia* commands and are not intended to be called directly.

The **ieee80211_watchdog**() function is intended to be called from a driver's *if_watchdog* routine. It is used to perform periodic cleanup of state within the software 802.11 stack, as well as timing out scans.

The **ieee80211_setmode**() function is called from within the 802.11 stack to change the mode of the driver's PHY; it is not intended to be called directly.

The **ieee80211_chan2mode**() function returns the PHY mode required for use with the channel *chan* on the device *ic*. This is typically used when selecting a rate set, to be advertised in beacons, for example.

The **ieee80211_rate2media**() function converts the bit rate *rate* (measured in units of 0.5Mbps) to an *ifmedia* sub-type, for the device *ic* running in PHY mode *mode*. The **ieee80211_media2rate**() performs the reverse of this conversion, returning the bit rate (in 0.5Mbps units) corresponding to an *ifmedia* sub-type.

SEE ALSO

`ieee80211_crypto`(9), `ieee80211_input`(9), `ieee80211_ioctl`(9), `ieee80211_node`(9), `ieee80211_output`(9), `ieee80211_proto`(9), `ieee80211_radiotap`(9)

HISTORY

The **ieee80211** series of functions first appeared in NetBSD 1.5, and were later ported to FreeBSD 4.6.

AUTHORS

This man page was written by Bruce M. Simpson ⟨bms@FreeBSD.org⟩ and Darron Broad ⟨darron@kewl.org⟩.

NAME

ieee80211_crypto_attach, **ieee80211_crypto_detach**, **ieee80211_crypto_encap** — 802.11 WEP encryption functions

SYNOPSIS

void
ieee80211_crypto_attach(*struct ieee80211com *ic*);

void
ieee80211_crypto_detach(*struct ieee80211com *ic*);

*struct ieee80211_key **
ieee80211_crypto_encap(*struct ieee80211com *ic*, *struct ieee80211_node *ni*, *struct mbuf *m0*);

DESCRIPTION

These functions provide encryption support for 802.11 device drivers.

The **ieee80211_crypto_attach**() function initializes crypto support for the interface *ic*. The default is null crypto.

The **ieee80211_crypto_detach**() function frees data structures associated with crypto support for the interface *ic*.

The two above functions are automatically called by the interface attach and detach routines, respectively.

The **ieee80211_crypto_encap**() function encapsulates the packet supplied in mbuf *m0*, with the crypto headers given the for node *ni*. Software encryption is possibly performed. In case of no specified key for *ni* or multicast traffic, the default key for the interface *ic* is used for encapsulation. The key is returned in the case of successful encapsulation, otherwise NULL is returned.

SEE ALSO

ieee80211(9)

HISTORY

The **ieee80211** series of functions first appeared in NetBSD 1.5, and were later ported to FreeBSD 4.6.

AUTHORS

This man page was written by Bruce M. Simpson ⟨bms@FreeBSD.org⟩ and Darron Broad ⟨darron@kewl.org⟩.

NAME

ieee80211_input, **ieee80211_decap**, **ieee80211_recv_mgmt** — software 802.11 stack input functions

SYNOPSIS

```
#include <net80211/ieee80211_var.h>
#include <net80211/ieee80211_proto.h>

void
ieee80211_input(struct ieee80211com *ic, struct mbuf *m,
    struct ieee80211_node *ni, int rssi, u_int32_t rstamp);

struct mbuf *
ieee80211_decap(struct ieee80211com *ic, struct mbuf *m);

void
ieee80211_recv_mgmt(struct ieee80211com *ic, struct mbuf *m0,
    struct ieee80211_node *ni, int subtype, int rssi, u_int32_t rstamp);
```

DESCRIPTION

These functions process received 802.11 frames.

The **ieee80211_input**() function takes an mbuf chain *m* containing a complete 802.11 frame from the driver *ic* and passes it to the software 802.11 stack for input processing. The *ni* argument specifies an instance of *struct ieee80211_node* (which may be driver-specific) representing the node from which the frame was received. The arguments *rssi* and *stamp* are typically derived from on-card data structures; they are used for recording the signal strength and time received of the frame respectively.

The **ieee80211_decap**() function performs decapsulation of the 802.11 frame in the mbuf chain *m* received by the device *ic*, taking the form of the 802.11 address fields into account; the structure of 802.11 addresses vary according to the intended source and destination of the frame. It is typically called from within **ieee80211_input**().

The **ieee80211_recv_mgmt**() performs input processing for 802.11 management frames. It is typically called from within **ieee80211_input**().

SEE ALSO

ieee80211(9)

HISTORY

The **ieee80211** series of functions first appeared in NetBSD 1.5, and were later ported to FreeBSD 4.6.

AUTHORS

This man page was written by Bruce M. Simpson ⟨bms@FreeBSD.org⟩ and Darron Broad ⟨darron@kewl.org⟩.

BUGS

There is no netisr queue specifically for the software 802.11 stack yet.

NAME

ieee80211_cfgget, **ieee80211_cfgset**, **ieee80211_ioctl** — 802.11 interface ioctl commands

SYNOPSIS

```
#include <net80211/ieee80211_var.h>
#include <net80211/ieee80211_proto.h>
#include <net80211/ieee80211_ioctl.h>

int
ieee80211_cfgget(struct ieee80211com *ic, u_long cmd, void *data);

int
ieee80211_cfgset(struct ieee80211com *ic, u_long cmd, void *data);

int
ieee80211_ioctl(struct ieee80211com *ic, u_long cmd, void *data);
```

DESCRIPTION

These functions are typically invoked by drivers in response to requests for information or to change settings from the userland.

The **ieee80211_cfgget**() and **ieee80211_cfgset**() functions implement a legacy interface for getting and setting 802.11 interface attributes respectively. The interface is compatible with the RIDs implemented by the wi(4) driver and used by the wiconfig(8) utility.

The **ieee80211_ioctl**() function implements ioctls such as key management for wireless devices. Ioctls related to the Ethernet layer also pass through here, but are handed off to **ether_ioctl**() when no match for *cmd* is found.

SEE ALSO

wi(4), ifconfig(8), wiconfig(8), ieee80211(9)

HISTORY

The **ieee80211** series of functions first appeared in NetBSD 1.5, and were later ported to FreeBSD 4.6.

AUTHORS

This man page was written by Bruce M. Simpson ⟨bms@FreeBSD.org⟩ and Darron Broad ⟨darron@kewl.org⟩.

NAME

ieee80211_node_attach, **ieee80211_node_lateattach**, **ieee80211_node_detach**, **ieee80211_begin_scan**, **ieee80211_next_scan**, **ieee80211_end_scan**, **ieee80211_create_ibss**, **ieee80211_alloc_node**, **ieee80211_dup_bss**, **ieee80211_find_node**, **ieee80211_free_node**, **ieee80211_free_allnodes**, **ieee80211_iterate_nodes** — software 802.11 stack node management functions

SYNOPSIS

```
#include <net80211/ieee80211_var.h>
#include <net80211/ieee80211_proto.h>
#include <net80211/ieee80211_node.h>

void
ieee80211_node_attach(struct ieee80211com *ic);

void
ieee80211_node_lateattach(struct ieee80211com *ic);

void
ieee80211_node_detach(struct ieee80211com *ic);

void
ieee80211_begin_scan(struct ieee80211com *ic, int reset);

void
ieee80211_next_scan(struct ieee80211com *ic);

void
ieee80211_end_scan(struct ieee80211com *ic);

void
ieee80211_create_ibss(struct ieee80211com *ic,
    struct ieee80211_channel *chan);

struct ieee80211_node *
ieee80211_alloc_node(struct ieee80211com *ic, u_int8_t *macaddr);

struct ieee80211_node *
ieee80211_dup_bss(struct ieee80211_node_table *nt,
    const u_int8_t *macaddr);

struct ieee80211_node *
ieee80211_find_node(struct ieee80211_node_table *nt,
    const u_int8_t *macaddr);

void
ieee80211_free_node(struct ieee80211_node *ni);

void
ieee80211_free_allnodes(struct ieee80211_node_table *nt);

void
ieee80211_iterate_nodes(struct ieee80211_node_table *nt,
    ieee80211_iter_func *f, void *arg);
```

DESCRIPTION

These functions are used to manage node lists within the software 802.11 stack. These lists are typically used for implementing host-mode AP functionality, or providing signal quality information about neighbour-

ing nodes.

The **ieee80211_node_attach**() function is called from ieee80211_ifattach(9) to initialize node database management callbacks for the interface *ic* (specifically for memory allocation, node copying and node signal inspection). These functions may be overridden in special circumstances, as long as this is done after calling ieee80211_ifattach(9) and prior to any other call which may allocate a node.

The **ieee80211_node_lateattach**() function initialises the *ic_bss* node element of the interface *ic* during ieee80211_media_init(9). This late attachment is to account for certain special cases described under **ieee80211_node_attach**().

The **ieee80211_node_detach**() function destroys all node database state associated with the interface *ic*, and is usually called during device detach.

The **ieee80211_begin_scan**() function initialises the node database in preparation of a scan for an access point on the interface *ic* and begins the scan. The parameter *reset* controls if a previously built node list should be cleared. The actual scanning for an access point is not fully automated: the device driver itself controls stepping through the channels, usually by a periodical callback.

The **ieee80211_next_scan**() function is used to inform the ieee80211(9) layer that the next channel for interface *ic* should be scanned.

The **ieee80211_create_ibss**() function sets up the net80211-specific portion of an interface's softc, *ic*, for use in IBSS mode.

The **ieee80211_end_scan**() function is called by **ieee80211_next_scan**() when the state machine has peformed a full cycle of scanning on all available radio channels. Internally, **ieee80211_end_scan**() will inspect the node cache associated with the interface *ic* for suitable access points found during scanning, and associate with one, should the parameters of the node match those of the configuration requested.

The **ieee80211_alloc_node**() function allocates an instance of *struct ieee80211_node* for a node having the MAC address *macaddr*, and associates it with the node table *nt*. If the allocation is successful, the node structure is initialised by **ieee80211_setup_node**(); otherwise, NULL is returned.

The **ieee80211_dup_bss**() function is similar to **ieee80211_alloc_node**(), but is instead used to create a node database entry for the BSSID *macaddr* associated with the note table *nt*. If the allocation is successful, the node structure is initialised by **ieee80211_setup_node**(); otherwise, NULL is returned.

The **ieee80211_find_node**() function will iterate through the node table *nt*, searching for a node entry which matches *macaddr*. If the entry is found, its reference count is incremented, and a pointer to the node is returned; otherwise, NULL is returned.

The **ieee80211_free_allnodes**() function will iterate through the node list calling **ieee80211_free_node**() for all the nodes in table *nt*.

The **ieee80211_iterate_nodes**() function will call the user-defined callback function *f* for all nodes in the table *nt*. The callback is invoked with the with the user-supplied value *arg* and a pointer to the current node.

SEE ALSO

ieee80211(9)

HISTORY

The **ieee80211** series of functions first appeared in NetBSD 1.5, and were later ported to FreeBSD 4.6.

AUTHORS

This man page was written by Bruce M. Simpson ⟨bms@FreeBSD.org⟩ and Darron Broad ⟨darron@kewl.org⟩.

NAME

ieee80211_encap, **ieee80211_add_rates**, **ieee80211_add_xrates**, **ieee80211_send_mgmt** — software 802.11 stack output functions

SYNOPSIS

```
#include <net80211/ieee80211_var.h>
#include <net80211/ieee80211_proto.h>

struct mbuf *
ieee80211_encap(struct ieee80211com *ic, struct mbuf *m,
    struct ieee80211_node *ni);

u_int8_t *
ieee80211_add_rates(u_int8_t *frm, const struct ieee80211_rateset *rs);

u_int8_t *
ieee80211_add_xrates(u_int8_t *frm, const struct ieee80211_rateset *rs);

int
ieee80211_send_mgmt(struct ieee80211com *ic, struct ieee80211_node *ni,
    int type, int arg);
```

DESCRIPTION

These functions handle the encapsulation and transmission of 802.11 frames within the software 802.11 stack.

The **ieee80211_encap**() function encapsulates an outbound data frame contained within the mbuf chain *m* from the interface *ic*. The argument *ni* is a reference to the destination node.

If the function is successful, the mbuf chain is updated with the 802.11 frame header prepended, and a pointer to the head of the chain is returned. If an error occurs, NULL is returned.

The **ieee80211_add_rates**() utility function is used to add the rate set element **rs* to the frame *frm*. A pointer to the location in the buffer after the addition of the rate set is returned. It is typically used when constructing management frames from within the software 802.11 stack.

The **ieee80211_add_xrates**() utility function is used to add the extended rate set element **rs* to the frame *frm*. A pointer to the location in the buffer after the addition of the rate set is returned. It is typically used when constructing management frames from within the software 802.11 stack in 802.11g mode.

The **ieee80211_send_mgmt**() function transmits a management frame on the interface *ic* to the destination node *ni* of type *type*.

The argument *arg* specifies either a sequence number for authentication operations, a status code for [re]association operations, or a reason for deauthentication and deassociation operations.

Nodes other than *ic_bss* have their reference count incremented to reflect their use for an indeterminate amount of time. This reference is freed when the function returns.

The function returns 0 if successful; if temporary buffer space is not available, the function returns ENOMEM.

SEE ALSO

ieee80211(9)

HISTORY

The **ieee80211** series of functions first appeared in NetBSD 1.5, and were later ported to FreeBSD 4.6.

AUTHORS

This man page was written by Bruce M. Simpson ⟨bms@FreeBSD.org⟩ and Darron Broad ⟨darron@kewl.org⟩.

NAME

ieee80211_proto_attach, **ieee80211_proto_detach**, **ieee80211_print_essid**, **ieee80211_dump_pkt**, **ieee80211_fix_rate**, **ieee80211_proto** — software 802.11 stack protocol helper functions

SYNOPSIS

```
#include <net80211/ieee80211_var.h>
#include <net80211/ieee80211_proto.h>

void
ieee80211_proto_attach(struct ieee80211com *ic);

void
ieee80211_proto_detach(struct ieee80211com *ic);

void
ieee80211_print_essid(u_int8_t *essid, int len);

void
ieee80211_dump_pkt(u_int8_t *buf, int len, int rate, int rssi);

int
ieee80211_fix_rate(struct ieee80211_node *ni, int flags);
```

DESCRIPTION

These functions are helper functions used throughout the software 802.11 protocol stack.

SEE ALSO

ieee80211(9)

HISTORY

The **ieee80211** series of functions first appeared in NetBSD 1.5, and were later ported to FreeBSD 4.6.

AUTHORS

This man page was written by Bruce M. Simpson ⟨bms@FreeBSD.org⟩ and Darron Broad ⟨darron@kewl.org⟩.

NAME

ieee80211_radiotap — software 802.11 stack packet capture definitions

SYNOPSIS

```
#include <net80211/ieee80211_var.h>
#include <net80211/ieee80211_ioctl.h>
#include <net80211/ieee80211_radiotap.h>
#include <net/bpf.h>
```

DESCRIPTION

The **ieee80211_radiotap** definitions provide a device-independent bpf(4) attachment for the capture of information about 802.11 traffic which is not part of the 802.11 frame structure.

Radiotap was designed to balance the desire for a capture format that conserved CPU and memory bandwidth on embedded systems, with the desire for a hardware-independent, extensible format that would support the diverse capabilities of virtually all 802.11 radios.

These considerations led radiotap to settle on a format consisting of a standard preamble followed by an extensible bitmap indicating the presence of optional capture fields.

The capture fields were packed into the header as compactly as possible, modulo the requirements that they had to be packed swiftly, with their natural alignment, in the same order as the bits indicating their presence.

This typically includes information such as signal quality and timestamps. This information may be used by a variety of user agents, including tcpdump(8). It is requested by using the bpf(4) data-link type DLT_IEEE_80211_RADIO.

Each frame using this attachment has the following header prepended to it:

```
struct ieee80211_radiotap_header {
        u_int8_t        it_version;     /* set to 0 */
        u_int8_t        it_pad;
        u_int16_t       it_len;         /* entire length */
        u_int32_t       it_present;     /* fields present */
} __attribute__((__packed__));
```

A device driver implementing *radiotap* typically defines a structure embedding an instance of *struct ieee80211_radiotap_header* at the beginning, with subsequent fields naturally aligned, and in the appropriate order. Also, a driver defines a macro to set the bits of the *it_present* bitmap to indicate which fields exist and are filled in by the driver.

Radiotap capture fields are in little-endian byte order.

Radiotap capture fields *must be naturally aligned*. That is, 16-, 32-, and 64-bit fields must begin on 16-, 32-, and 64-bit boundaries, respectively. In this way, drivers can avoid unaligned accesses to radiotap capture fields. radiotap-compliant drivers must insert padding before a capture field to ensure its natural alignment. radiotap-compliant packet dissectors, such as tcpdump(8), expect the padding.

Developers beware: all compilers may not pack structs alike. If a driver developer constructs their radiotap header with a packed structure, in order to ensure natural alignment, then it is important that they insert padding bytes by themselves.

Radiotap headers are copied to the userland via a separate bpf attachment. It is necessary for the driver to create this attachment after calling ieee80211_ifattach(9) by calling **bpfattach2**() with the data-link type set to DLT_IEEE802_11_RADIO.

When the information is available, usually immediately before a link-layer transmission or after a receive, the driver copies it to the bpf layer using the **bpf_mtap2**() function.

The following extension fields are defined for *radiotap*, in the order in which they should appear in the buffer copied to userland:

IEEE80211_RADIOTAP_TSFT
: This field contains the unsigned 64-bit value, in microseconds, of the MAC's 802.11 Time Synchronization Function timer, when the first bit of the MPDU arrived at the MAC. This field should be present for received frames only.

IEEE80211_RADIOTAP_FLAGS
: This field contains a single unsigned 8-bit value, containing a bitmap of flags specifying properties of the frame being transmitted or received.

IEEE80211_RADIOTAP_RATE
: This field contains a single unsigned 8-bit value, which is the data rate in use in units of 500Kbps.

IEEE80211_RADIOTAP_CHANNEL
: This field contains two unsigned 16-bit values. The first value is the frequency upon which this PDU was transmitted or received. The second value is a bitmap containing flags which specify properties of the channel in use. These are documented within the header file, <net80211/ieee80211_radiotap.h>.

IEEE80211_RADIOTAP_FHSS
: This field contains two 8-bit values. This field should be present for frequency-hopping radios only. The first byte is the hop set. The second byte is the pattern in use.

IEEE80211_RADIOTAP_DBM_ANTSIGNAL
: This field contains a single signed 8-bit value, which indicates the RF signal power at the antenna, in decibels difference from 1mW.

IEEE80211_RADIOTAP_DBM_ANTNOISE
: This field contains a single signed 8-bit value, which indicates the RF noise power at the antenna, in decibels difference from 1mW.

IEEE80211_RADIOTAP_LOCK_QUALITY
: This field contains a single unsigned 16-bit value, indicating the quality of the Barker Code lock. No unit is specified for this field. There does not appear to be a standard way of measuring this at this time; this quantity is often referred to as "Signal Quality" in some datasheets.

IEEE80211_RADIOTAP_TX_ATTENUATION
: This field contains a single unsigned 16-bit value, expressing transmit power as unitless distance from maximum power set at factory calibration. 0 indicates maximum transmit power. Monotonically nondecreasing with lower power levels.

IEEE80211_RADIOTAP_DB_TX_ATTENUATION
: This field contains a single unsigned 16-bit value, expressing transmit power as decibel distance from maximum power set at factory calibration. 0 indicates maximum transmit power. Monotonically nondecreasing with lower power levels.

IEEE80211_RADIOTAP_DBM_TX_POWER
: Transmit power expressed as decibels from a 1mW reference. This field is a single signed 8-bit value. This is the absolute power level measured at the antenna port.

IEEE80211_RADIOTAP_ANTENNA
: For radios which support antenna diversity, this field contains a single unsigned 8-bit value specifying which antenna is being used to transmit or receive this frame. The first antenna is antenna 0.

`IEEE80211_RADIOTAP_DB_ANTSIGNAL`
: This field contains a single unsigned 8-bit value, which indicates the RF signal power at the antenna, in decibels difference from an arbitrary, fixed reference.

`IEEE80211_RADIOTAP_DB_ANTNOISE`
: This field contains a single unsigned 8-bit value, which indicates the RF noise power at the antenna, in decibels difference from an arbitrary, fixed reference.

`IEEE80211_RADIOTAP_RX_FLAGS`
: An unsigned 16-bit bitmap indicating properties of received frames.

`IEEE80211_RADIOTAP_TX_FLAGS`
: An unsigned 16-bit bitmap indicating properties of transmitted frames.

`IEEE80211_RADIOTAP_RTS_RETRIES u_int8_t data`
: Unsigned 8-bit value indicating how many times the NIC retransmitted the Request to Send (RTS) in an RTS/CTS handshake before receiving an 802.11 Clear to Send (CTS).

`IEEE80211_RADIOTAP_DATA_RETRIES`
: Unsigned 8-bit value indicating how many times the NIC retransmitted a unicast data packet before receiving an 802.11 Acknowledgement.

`IEEE80211_RADIOTAP_EXT`
: This bit is reserved for any future extensions to the *`radiotap`* structure. A driver sets `IEEE80211_RADIOTAP_EXT` to extend the it_present bitmap by another 64 bits. The bitmap can be extended by multiples of 32 bits to 96, 128, 160 bits or longer, by setting `IEEE80211_RADIOTAP_EXT` in the extensions. The bitmap ends at the first extension field where `IEEE80211_RADIOTAP_EXT` is not set.

EXAMPLES

Radiotap header for the Cisco Aironet driver:

```
struct an_rx_radiotap_header {
        struct ieee80211_radiotap_header      ar_ihdr;
        u_int8_t        ar_flags;
        u_int8_t        ar_rate;
        u_int16_t       ar_chan_freq;
        u_int16_t       ar_chan_flags;
        u_int8_t        ar_antsignal;
        u_int8_t        ar_antnoise;
} __attribute__((__packed__));
```

Bitmap indicating which fields are present in the above structure:

```
#define AN_RX_RADIOTAP_PRESENT \
        ((1 >> IEEE80211_RADIOTAP_FLAGS) | \
         (1 >> IEEE80211_RADIOTAP_RATE) | \
         (1 >> IEEE80211_RADIOTAP_CHANNEL) | \
         (1 >> IEEE80211_RADIOTAP_DBM_ANTSIGNAL) | \
         (1 >> IEEE80211_RADIOTAP_DBM_ANTNOISE))
```

SEE ALSO

`bpf`(4), `ieee80211`(9)

HISTORY

The **ieee80211_radiotap** definitions first appeared in NetBSD 1.5, and were later ported to FreeBSD 4.6.

AUTHORS

The **ieee80211_radiotap** interface was designed and implemented by David Young ⟨dyoung@pobox.com⟩. David Young is the maintainer of the radiotap capture format. Contact him to add new capture fields.

This manual page was written by Bruce M. Simpson ⟨bms@FreeBSD.org⟩ and Darron Broad ⟨darron@kewl.org⟩.

NAME

iic_acquire_bus, **iic_release_bus**, **iic_exec**, **iic_smbus_write_byte**, **iic_smbus_read_byte**, **iic_smbus_receive_byte** — Inter IC (I2C) bus

SYNOPSIS

```
#include <dev/i2c/i2cvar.h>

int
iic_acquire_bus(i2c_tag_t ic, int flags);

int
iic_release_bus(i2c_tag_t ic, int flags);

int
iic_exec(i2c_tag_t ic, i2c_op_t op, i2c_addr_t addr, const void *cmdbuf,
    size_t cmdlen, void *buf, size_t len, int flags);

int
iic_smbus_write_byte(i2c_tag_t ic, i2c_addr_t addr, uint8_t cmd,
    uint8_t data, int flags);

int
iic_smbus_read_byte(i2c_tag_t ic, i2c_addr_t addr, uint8_t cmd,
    uint8_t *datap, int flags);

int
iic_smbus_receive_byte(i2c_tag_t ic, i2c_addr_t addr, uint8_t *datap,
    int flags);
```

DESCRIPTION

I2C is a two-wire bus developed by Philips used for connecting integrated circuits. It is commonly used for connecting devices such as EEPROMs, temperature sensors, fan controllers, real-time clocks, tuners, and other types of integrated circuits. The **iic** interface provides a means of communicating with I2C-connected devices. The System Management Bus, or SMBus, is a variant of the I2C bus with a simplified command protocol and some electrical differences.

DATA TYPES

Drivers for devices attached to the I2C bus will make use of the following data types:

i2c_tag_t Controller tag for the I2C bus. This is a pointer to a *struct i2c_controller*, consisting of function pointers filled in by the I2C controller driver.

i2c_op_t I2C bus operation. The following I2C bus operations are defined:

I2C_OP_READ
: Perform a read operation.

I2C_OP_READ_WITH_STOP
: Perform a read operation and send a STOP condition on the I2C bus at the conclusion of the read.

I2C_OP_WRITE
: Perform a write operation.

I2C_OP_WRITE_WITH_STOP
: Perform a write operation and send a STOP condition on the I2C bus at the conclusion of the write.

i2c_addr_t I2C device address.

struct i2c_attach_args
Devices are attached to an I2C bus using this structure. The structure is defined as follows:

```
struct i2c_attach_args {
        i2c_tag_t ia_tag;       /* controller */
        i2c_addr_t ia_addr;     /* address of device */
        int ia_size;            /* size (for EEPROMs) */
};
```

FUNCTIONS

The following functions comprise the API provided to drivers of I2C-connected devices:

iic_acquire_bus(*ic*, *flags*)
Acquire an exclusive lock on the I2C bus. This is required since only one device may communicate on the I2C bus at a time. Drivers should acquire the bus lock, perform the I2C bus operations necessary, and then release the bus lock. Passing the I2C_F_POLL flag indicates to **iic_acquire_bus**() that sleeping is not permitted.

iic_release_bus(*ic*, *flags*)
Release an exclusive lock on the I2C bus. If the I2C_F_POLL flag was passed to **iic_acquire_bus**(), it must also be passed to **iic_release_bus**().

iic_exec(*ic*, *op*, *addr*, *cmdbuf*, *cmdlen*, *buf*, *len*, *flags*)
Perform a series of I2C transactions on the bus. **iic_exec**() initiates the operation by sending a START condition on the I2C bus and then transmitting the address of the target device along with the transaction type. If *cmdlen* is non-zero, the command pointed to by *cmdbuf* is then sent to the device. If *buflen* is non-zero, **iic_exec**() will then transmit or receive the data, as indicated by *op*. If *op* indicates a read operation, **iic_exec**() will send a REPEATED START before transferring the data. If *op* so indicates, a STOP condition will be sent on the I2C bus at the conclusion of the operation. Passing the I2C_F_POLL flag indicates to **iic_exec**() that sleeping is not permitted.

iic_smbus_write_byte(*ic*, *addr*, *cmd*, *data*, *flags*)
Perform an SMBus WRITE BYTE operation. This is equivalent to I2C_OP_WRITE_WITH_STOP with *cmdlen* of 1 and *len* of 1.

iic_smbus_read_byte(*ic*, *addr*, *cmd*, *datap*, *flags*)
Perform an SMBus READ BYTE operation. This is equivalent to I2C_OP_READ_WITH_STOP with *cmdlen* of 1 and *len* of 1.

iic_smbus_receive_byte(*ic*, *addr*, *datap*, *flags*)
Perform an SMBus RECEIVE BYTE operation. This is equivalent to I2C_OP_READ_WITH_STOP with *cmdlen* of 0 and *len* of 1.

CONTROLLER INTERFACE

The I2C controller driver must fill in the function pointers of an *i2c_controller* structure, which is defined as follows:

```
struct i2c_controller {
        void    *ic_cookie;     /* controller private */

        int     (*ic_acquire_bus)(void *, int);
        void    (*ic_release_bus)(void *, int);
```

```
        int     (*ic_exec)(void *, i2c_op_t, i2c_addr_t,
                   const void *, size_t, void *, size_t, int);

        int     (*ic_send_start)(void *, int);
        int     (*ic_send_stop)(void *, int);
        int     (*ic_initiate_xfer)(void *, i2c_addr_t, int);
        int     (*ic_read_byte)(void *, uint8_t *, int);
        int     (*ic_write_byte)(void *, uint8_t, int);
};
```

The **(*ic_acquire_bus)**() and **(*ic_release_bus)**() functions must always be provided.

The controller driver may elect to provide an **(*ic_exec)**() function. This function is intended for use by automated controllers that do not provide manual control over I2C bus conditions such as START and STOP.

If the **(*ic_exec)**() function is not provided, the following 5 functions will be used by **iic_exec**() in order to execute the I2C bus operation:

(*ic_send_start)(*cookie*, *flags*)
: Send a START condition on the I2C bus. The `I2C_F_POLL` flag indicates that sleeping is not permitted.

(*ic_send_stop)(*cookie*, *flags*)
: Send a STOP condition on the I2C bus. The `I2C_F_POLL` flag indicates that sleeping is not permitted.

(*ic_initiate_xfer)(*cookie*, *addr*, *flags*)
: Initiate a transfer on the I2C bus by sending a START condition and then transmitting the I2C device address and transfer type. The `I2C_F_READ` flag indicates a read transfer; the lack of this flag indicates a write transfer. The `I2C_F_POLL` flag indicates that sleeping is not permitted. The error code `ETIMEDOUT` should be returned if a timeout that would indicate that the device is not present occurs.

(*ic_read_byte)(*cookie*, *datap*, *flags*)
: Read a byte from the I2C bus into the memory location referenced by *datap*. The `I2C_F_LAST` flag indicates that this is the final byte of the transfer, and that a NACK condition should be sent on the I2C bus following the transfer of the byte. The `I2C_F_STOP` flag indicates that a STOP condition should be sent on the I2C bus following the transfer of the byte. The `I2C_F_POLL` flag indicates that sleeping is not permitted.

(*ic_write_byte)(*cookie*, *data*, *flags*)
: Write the byte contained in *data* to the I2C bus. The `I2C_F_STOP` flag indicates that a STOP condition should be sent on the I2C bus following the transfer of the byte. The `I2C_F_POLL` flag indicates that sleeping is not permitted.

SEE ALSO

`iic`(4)

HISTORY

The **iic** API first appeared in NetBSD 2.0. OpenBSD support was added in OpenBSD 3.6.

AUTHORS

The **iic** API was written by Steve C. Woodford and Jason R. Thorpe for NetBSD and then ported to OpenBSD by Alexander Yurchenko ⟨grange@openbsd.org⟩.

NAME

imax, **imin**, **lmax**, **lmin**, **max**, **min**, **ulmax**, **ulmin** — compare integers

SYNOPSIS

int
imax(*int a*, *int b*);

int
imin(*int a*, *int b*);

long
lmax(*long a*, *long b*);

long
lmin(*long a*, *long b*);

u_int
max(*u_int a*, *u_int b*);

u_int
min(*u_int a*, *u_int b*);

u_long
ulmax(*u_long a*, *u_long b*);

u_long
ulmin(*u_long a*, *u_long b*);

DESCRIPTION

The **imin**(), **lmin**(), **min**(), and **ulmin**() functions return whichever argument is algebraically smaller, differing only in their argument and return types: these functions operate on, respectively, natural size, long, unsigned and unsigned long integers.

The **imax**(), **lmax**(), **max**(), and **ulmax**() functions are identical except that they return the algebraically larger argument between *a* and *b*.

NAME

in_cksum, **in4_cksum**, **in6_cksum** — compute Internet checksum

SYNOPSIS

```
uint16_t
in_cksum(struct mbuf *m, int len);

uint16_t
in4_cksum(struct mbuf *m, uint8_t nxt, int off, int len);

uint16_t
in6_cksum(struct mbuf *m, uint8_t nxt, int off, int len);
```

DESCRIPTION

These functions are used to compute the ones-complement checksum required by IP and IPv6. The **in4_cksum**() function is used to compute the transport-layer checksum required by tcp(4) and udp(4) over a range of bytes starting at *off* and continuing on for *len* bytes within the mbuf *m*.

If the *nxt* parameter is non-zero, it is assumed to be an IP protocol number. It is also assumed that the data within *m* starts with an IP header, and the transport-layer header starts at *off*; a pseudo-header is constructed as specified in RFC768 and RFC793, and the pseudo-header is prepended to the data covered by the checksum.

The **in6_cksum**() function is similar; if *nxt* is non-zero, it is assumed that *m* starts with an IPv6 header, and that the transport-layer header starts after *off* bytes.

The **in_cksum**() function is equivalent to **in4_cksum**(*m*, *0*, *0*, *len*).

These functions are always performance critical and should be reimplemented in assembler or optimized C for each platform; when available, use of repeated full-width add-with-carry followed by reduction of the sum to a 16 bit width usually leads to best results. See RFC's 1071, 1141, 1624, and 1936 for more information about efficient computation of the internet checksum.

RETURN VALUES

All three functions return the computed checksum value.

SEE ALSO

inet(4), inet6(4), tcp(4), udp(4), protocols(5), mbuf(9)

STANDARDS

These functions implement the Internet transport-layer checksum as specified in RFC768, RFC793, and RFC2460.

BUGS

The **in6_cksum**() function currently requires special handling of link-local addresses in the pseudo-header due to the use of embedded scope-id's within link-local addresses.

NAME

in_getifa — Look up the IPv4 source address best matching an IPv4 destination

SYNOPSIS

```
options IPSELSRC
#include <netinet/in_selsrc.h>

struct ifaddr *
in_getifa(struct ifaddr *ifa, const struct sockaddr *dst0);
```

DESCRIPTION

in_getifa enforces the IPv4 source-address selection policy. Add the source-address selection policy mechanism to your kernel with **options IPSELSRC**. **options IPSELSRC** lets the operator set the policy for choosing the source address of any socket bound to the "wildcard" address, INADDR_ANY. Note that the policy is applied *after* the kernel makes its forwarding decision, thereby choosing the output interface; in other words, this mechanism does not affect whether or not NetBSD is a "strong ES".

An operator affects the source-address selection using sysctl(8) and ifconfig(8). Operators set policies with sysctl(8). Some policies consider the "preference number" of an address. An operator may set preference numbers for each address with ifconfig(8).

A source-address policy is a priority-ordered list of source-address ranking functions. A ranking function maps its arguments, (*source address*, *source index*, *source preference*, *destination address*), to integers. The *source index* is the position of *source address* in the interface address list; the index of the first address is 0. The *source preference* is the preference number the operator assigned to *source address*. The *destination address* is the socket peer / packet destination.

Presently, there are four ranking functions to choose from:

index — ranks by *source index*; lower indices are ranked more highly.

preference — ranks by *source preference*; higher preference numbers are ranked more highly.

common-prefix-len — ranks each *source address* by the length of the longest prefix it has in common with *destination address*; longer common prefixes rank more highly.

same-category — determines the "categories" of *source* and *destination address*. A category is one of *private*, *link-local*, or *other*. If the categories exactly match, same-category assigns a rank of 2. Some sources are ranked 1 by category: a *link-local* source with a *private* destination, a *private* source with a *link-local* destination, and a *private* source with an *other* destination rank 1. All other sources rank 0.

Categories are defined as follows.

private	RFC1918 networks, 192.168/16, 172.16/12, and 10/8
link-local	169.254/16, 224/24
other	all other networks---i.e., not private, not link-local

To apply a policy, the kernel applies all ranking functions in the policy to every source address, producing a vector of ranks for each source. The kernel sorts the sources in descending, lexicographical order by their rank-vector, and chooses the highest-ranking (first) source. The kernel breaks ties by choosing the source with the least *source index*.

The operator may set a policy on individual interfaces. The operator may also set a global policy that applies to all interfaces whose policy he does not set individually.

Here is the sysctl tree for the policy at system startup:

```
net.inet.ip.selectsrc.default = index
net.inet.ip.interfaces.ath0.selectsrc =
net.inet.ip.interfaces.sip0.selectsrc =
net.inet.ip.interfaces.sip1.selectsrc =
net.inet.ip.interfaces.lo0.selectsrc =
net.inet.ip.interfaces.pflog0.selectsrc =
```

The policy on every interface is the "empty" policy, so the default policy applies. The default policy, *index*, is the "historical" policy in NetBSD.

The operator may override the default policy on ath0,

```
# sysctl -w net.inet.ip.interfaces.ath0.selectsrc=same-category,common-pr
```

yielding this policy:

```
net.inet.ip.selectsrc.default = index
net.inet.ip.interfaces.ath0.selectsrc = same-category,common-prefix-len,preferen
```

The operator may set a new default,

```
# sysctl -w net.inet.ip.selectsrc.debug=> same-category,common-prefix-len,prefer
# sysctl -w net.inet.ip.interfaces.ath0.selectsrc=
```

yielding this policy:

```
net.inet.ip.selectsrc.default = same-category,common-prefix-len,preference
net.inet.ip.interfaces.ath0.selectsrc =
```

In a number of applications, the policy above will usually pick suitable source addresses if ath0 is configured in this way:

```
# ifconfig ath0 inet 64.198.255.1/24
# ifconfig ath0 inet 10.0.0.1/24
# ifconfig ath0 inet 169.254.1.1/24
# ifconfig ath0 inet 192.168.49.1/24 preference 5
# ifconfig ath0 inet 192.168.37.1/24 preference 9
```

A sysctl, net.inet.ip.selectsrc.debug, turns on and off debug messages concerned with source selection. You may set it to 0 (no messages) or 1.

SEE ALSO

`ifconfig`(8), `sysctl`(8)

STANDARDS

The family of IPv6 source-address selection policies defined by `RFC3484` resembles the family of IPv4 policies that **`in_getifa`** enforces.

AUTHORS

David Young ⟨dyoung@NetBSD.org⟩

BUGS

With **`options IPSELSRC`**, a new interface `ioctl`(2), `SIOCSIFADDRPREF`, was introduced. It ought to be documented in `inet`(4). Also, `options`(4) ought to cross-reference this manual page.

This work should be used to set IPv6 source-address selection policies, especially the family of policies defined by `RFC3484`.

NAME

inittodr — initialize system time

SYNOPSIS

void
inittodr(*time_t base*);

DESCRIPTION

The **inittodr**() function determines the time and sets the system clock. It tries to pick the correct time using a set of heuristics that examine the system's battery-backed clock and the time reported by the file system, as given in *base*. Those heuristics include:

- If the battery-backed clock has a valid time, and is not significantly behind the time provided by *base*, it is used.
- If the battery-backed clock does not have a valid time, or is significantly behind the time provided in *base*, and the time provided in *base* is within reason, *base* is used as the current time.
- If the battery-backed clock appears invalid, and *base* appears non-sensical or was not provided (was given as zero), an arbitrary base (typically some time within the same year that the kernel was last updated) will be used.

Once a system time has been determined, it is stored in the *time* variable.

DIAGNOSTICS

The **inittodr**() function prints diagnostic messages if it has trouble figuring out the system time. Conditions that can cause diagnostic messages to be printed include:

- There is no battery-backed clock present on the system.
- The battery-backed clock's time appears nonsensical.
- The *base* time appears nonsensical.
- The *base* time and the battery-backed clock's time differ by a large amount.

SEE ALSO

clock_ymdhms_to_secs(9), resettodr(9), time_second(9)

BUGS

Some systems use heuristics for picking the correct time that are slightly different.

NAME

IOASIC, **ioasic_intr_establish**, **ioasic_intr_disestablish**, **ioasic_intr_evcnt**, **ioasic_attach_devs**, **ioasic_submatch** — baseboard I/O control ASIC for DEC TURBOchannel systems

SYNOPSIS

```
#include <machine/bus.h>
#include <dev/tc/tcvar.h>
#include <dev/tc/ioasicreg.h>
#include <dev/tc/ioasicvar.h>

void
ioasic_intr_establish(struct device *dev, void *cookie, int level,
    int (*handler)(void *), void *arg);

void
ioasic_intr_disestablish(struct device *dev, void *cookie);

const struct evcnt *
ioasic_intr_evcnt(struct device *dev, void *cookie);

void
ioasic_attach_devs(struct ioasic_softc *sc,
    struct ioasic_dev *ioasic_devs, int ioasic_ndevs);

int
ioasic_submatch(struct cfdata *match, struct ioasicdev_attach_args *ia);
```

DESCRIPTION

The **IOASIC** device provides support for the DEC proprietary IOCTL ASIC found on all DEC TURBOchannel machines with MIPS (DECstation 5000 series, excluding the 5000/200) and Alpha (3000-series) systems. The **IOASIC** is memory-mapped into the TURBOchannel system slot to interface up to sixteen I/O devices. It connects the TURBOchannel to a 16-bit wide I/O bus and supplies various control signals to the devices that share this bus.

The **IOASIC** provides hardware DMA channels and interrupt support. DMA transfers are between one and four 32-bit words (16 bytes) in length, depending on the device. The **IOASIC** stores the data in internal data registers. The data is transferred to and from the registers in 16-bit words to the device. Various interrupts are signalled on DMA pointer-related conditions.

DATA TYPES

Drivers for devices attached to the **IOASIC** will make use of the following data types:

struct ioasicdev_attach_args

A structure used to inform the driver of the **IOASIC** device properties. It contains the following members:

```
char                    iada_modname
tc_offset_t             iada_offset
tc_addr_t               iada_addr
void                    *iada_cookie;
```

struct ioasic_softc

The parent structure which contains at the following members which are useful for drivers:

```
bus_space_tag_t                 sc_bst;
bus_space_handle_t      sc_bsh;
bus_dma_tag_t           sc_dmat;
```

struct ioasic_dev

A structure describing the machine-dependent devices attached to the **IOASIC** containing the following members:

```
char                    *iad_modname;
tc_offset_t             iad_offset;
void                    *iad_cookie;
uint32_t                iad_intrbits;
```

FUNCTIONS

ioasic_intr_establish(*dev*, *cookie*, *level*, *handler*, *arg*)

Establish an interrupt handler with device *dev* for the interrupt described completely by *cookie*. The priority of the interrupt is specified by *level*. When the interrupt occurs the function *handler* is called with argument *arg*.

ioasic_intr_disestablish(*dev*, *cookie*)

Dis-establish the interrupt handler with device *dev* for the interrupt described complete ly *cookie*.

ioasic_intr_evcnt(*dev*, *cookie*)

Do interrupt event counting with device *dev* for the event described completely by *cookie*.

ioasic_attach_devs(*sc*, *ioasic_devs*, *ioasic_ndevs*)

Configure each of the *ioasic_ndevs* devices in *ioasic_devs*.

ioasic_submatch(*match*, *ia*)

Check that the device offset is not OASIC_OFFSET_UNKNOWN.

The **ioasic_intr_establish**(), **ioasic_intr_disestablish**(), and **ioasic_intr_evcnt**() functions are likely to used by all **IOASIC** device drivers. The **ioasic_attach_devs**() function is used by ioasic driver internally and is of interest to driver writers because it must be aware of your device for it to be found during autoconfiguration.

AUTOCONFIGURATION

The IOASIC is a direct-connection bus. During autoconfiguration, machine-dependent code will provide an array of *struct ioasic_devs* describing devices attached to the **IOASIC** to be used by the ioasic driver. The ioasic driver will pass this array to **ioasic_attach_devs**() to attach the drivers with the devices.

Drivers match the device using *iada_modname*.

During attach, all drivers should use the parent's bus_space and bus_dma resources, and map the appropriate bus_space region using **bus_space_subregion**() with *iada_offset*.

DMA SUPPORT

No additional support is provided for **IOASIC** DMA beyond the facilities provided by the bus_dma(9) interface.

The **IOASIC** provides two pairs of DMA address pointers (transmitting and receiving) for each DMA-capable device. The pair of address pointers point to consecutive (but not necessarily contiguous) DMA blocks of size IOASIC_DMA_BLOCKSIZE. Upon successful transfer of the first block, DMA continues to the next block and an interrupt is posted to signal an address pointer update. DMA transfers are enabled and disabled by bits inside the **IOASIC** status (CSR) register.

The interrupt handler must update the address pointers to point to the next block in the DMA transfer. The address pointer update must be completed before the completion of the second DMA block, otherwise a DMA overrun error condition will occur.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent IOASIC subsystem can be found. All pathnames are relative to `/usr/src`.

The IOASIC subsystem itself is implemented within the file `sys/dev/tc/ioasic_subr.c`. Machine-dependent portions can be found in `sys/arch/<arch>/tc/ioasic.c`.

SEE ALSO

`ioasic`(4), `autoconf`(9), `bus_dma`(9), `bus_space`(9), `driver`(9)

NAME

ioctl — how to implement a new ioctl call to access device drivers

SYNOPSIS

```
#include <sys/ioctl.h>
#include <sys/ioccom.h>

int
ioctl(int, unsigned long, ...);
```

DESCRIPTION

ioctl are internally defined as

#define FOOIOCTL fun(t,n,pt)

where the different variables and functions are:

FOOIOCTL the name which will later be given in the ioctl(2) system call as second argument, e.g., `ioctl(s, FOOIOCTL, ...)`.

fun() a macro which can be one of

- _IO the call is a simple message to the kernel by itself. It does not copy anything into the kernel, nor does it want anything back.
- _IOR the call only reads parameters from the kernel and does not pass any to it
- _IOW the call only writes parameters to the kernel, but does not want anything back
- _IOWR the call writes data to the kernel and wants information back.

t This integer describes to which subsystem the ioctl applies. *t* can be one of

- '1' pulse-per-second interface
- '4' isdn(4)
- 'a' ISO networking
- 'A' ac devices (hp300)
- 'A' Advanced Power Management (hpcmips, i386, sparc), see apm(4)
- 'A' ADB devices (mac68k, macppc)
- 'A' audio(4)
- 'A' isdntel(4)
- 'b' tb(4)
- 'b' Bluetooth HCI sockets, see bluetooth(4)
- 'b' Bluetooth Hub Control, see bthub(4)
- 'b' Bluetooth SCO audio driver, see btsco(4)
- 'B' bell device (x68k)
- 'B' bpf(4)
- 'c' coda
- 'c' cd(4)
- 'c' ch(4)
- 'C' clock devices (amiga, atari, hp300, x68k)
- 'C' isdnctl(4)
- 'd' the disk subsystem
- 'E' envsys(4)

'f'	files
'F'	Sun-compatible framebuffers
'F'	`ccd`(4) and `vnd`(4)
'g'	qdss framebuffers
'G'	grf devices (amiga, atari, hp300, mac68k, x68k)
'h'	HIL devices (hp300)
'H'	HIL devices (hp300)
'H'	HPc framebuffers
'i'	a (pseudo) interface
'I'	`ite`(4) (mac68k)
'J'	ISA joystick interface
'k'	Sun-compatible (and other) keyboards
'K'	`lkm`(4)
'l'	leo devices (atari)
'm'	`mtio`(4)
'M'	mouse devices (atari)
'M'	`mlx`(4)
'n'	virtual console device (arm32)
'n'	SMB networking
'O'	OpenPROM and OpenFirmware
'p'	power control (x68k)
'P'	parallel port (amiga, x68k)
'P'	profiling (arm32)
'P'	printer/plotter interface (hp300)
'P'	`magma`(4) bpp (sparc)
'q'	`altq`(9)
'q'	pmax graphics devices
'Q'	`altq`(9)
'Q'	raw SCSI commands
'r'	the routing subsystem
'r'	`md`(4)
'R'	`isdnbchan`(4)
'R'	`rnd`(4)
's'	the socket layer
's'	satlink devices
'S'	SCSI disks (arc, hp300, pmax)
'S'	watchdog devices (sh3)
'S'	ISA speaker devices
'S'	stic devices
'S'	scanners
't'	the tty layer
'u'	user defined ???
'U'	scsibus (see `scsi`(4))
'v'	Sun-compatible "firm events"
'V'	view device (amiga, atari)
'V'	sram device (x68k)
'w'	watchdog devices
'W'	wt devices
'W'	wscons devices

'x' bt8xx devices
'Z' ite devices (amiga, atari, x68k)
'Z' passthrough ioctls

n This numbers the ioctl within the group. There may be only one *n* for a given *t*. This is a unsigned 8 bit number.

pt This specifies the type of the passed parameter. This one gets internally transformed to the size of the parameter, so for example, if you want to pass a structure, then you have to specify that structure and not a pointer to it or sizeof(struct foo)

In order for the new ioctl to be known to the system it is installed in either <sys/ioctl.h> or one of the files that are reached from <sys/ioctl.h>.

EXAMPLES

```
#define FOOIOCTL        _IOWR('i', 23, int)

int a = 3;
error = ioctl(s, FOOICTL, &a);
```

Within the ioctl()-routine of the driver, it can be then accessed like

```
driver_ioctl(..., u_long cmd, void *data)
{
        ...
        switch (cmd) {

        case FOOIOCTL:
                int *a = (int *)data;
                printf(" Value passed: %d\n", *a);
                break;
        }
}
```

NOTES

Note that if you for example try to read information from an ethernet driver where the name of the card is included in the third argument (e.g., ioctl(s, READFROMETH, struct ifreq *)), then you have to use the _IOWR() form not the _IOR(), as passing the name of the card to the kernel already consists of writing data.

RETURN VALUES

All ioctl() routines should return either 0 or a defined error code. The use of magic numbers such as -1, to indicate that a given ioctl code was not handled is strongly discouraged. The value -1 coincides with the historic value for **ERESTART** which was shown to produce user space code that never returned from a call to `ioctl`(2).

For ioctl codes that are not handled by a given routine, the pseudo error value **EPASSTHROUGH** is provided. **EPASSTHROUGH** indicates that no error occurred during processing (it did not fail), but neither was anything processed (it did not succeed). This supersedes the use of either **ENOTTY** (which is an explicit failure) or -1 (which has no contextual meaning) as a return value. **ENOTTY** will get passed directly back to user space and bypass any further processing by other ioctl layers. Only code that wishes to suppress possible further processing of an ioctl code (e.g., the tty line discipline code) should return **ENOTTY**. All other code should return **EPASSTHROUGH**, even if it knows that no other layers will be called upon.

If the value **EPASSTHROUGH** is returned to **sys_ioctl**(), then it will there be changed to **ENOTTY** to be returned to user space, thereby providing the proper error notification to the application.

SEE ALSO

`ioctl`(2)

NAME

ipkdb — machine-dependent interface to ipkdb

SYNOPSIS

```
#include <ipkdb/ipkdb.h>

void
ipkdb_init(void);

void
ipkdb_connect(int when);

int
ipkdbcmds(void);

void
ipkdbinit(void);

void
ipkdb_trap(void);

int
ipkdb_poll(void);

int
ipkdbif_init(struct ipkdb_if *kip);

int
ipkdbfbyte(u_char *c);

int
ipkdbsbyte(u_char *c, int i);
```

DESCRIPTION

The machine-dependent code must support this interface for operation with ipkdb(4).

During system bootstrap, machine-dependent code must invoke **ipkdb_init**(). If the kernel is booted with RB_KDB set in boothowto, then ipkdb(4) is enabled by invoking **ipkdb_connect**(), setting the *when* argument to 0.

ipkdbcmds() is invoked by machine-dependent code when the trap mechanism determines that the debugger should be entered, i.e., on a single step or breakpoint interrupt from kernel code. The trapping mechanism should already have stored the registers into the global area ipkdbregs. The layout of this area must be the same as that expected by gdb(1). Valid return values are:

IPKDB_CMD_RUN	user wants to continue
IPKDB_CMD_STEP	user wants to do single stepping
IPKDB_CMD_EXIT	user has detached from debugging

FUNCTIONS

The machine-dependent code must provide the following functions for the machine-independent code.

ipkdbinit() This routine gets called when the debugger should be entered for the first time.

ipkdb_trap() This routine is part of the trap handler. Whenever a trap happens (e.g., when hitting a breakpoint during debugging), **ipkdb_trap**() decides if the Debugger needs to be called. If there are other ways to decide that, it's not necessary to provide an **ipkdb_trap**() implementation.

ipkdb_poll() This routine gets called after a panic to check for a key press by the user. If implemented it allows the user to press any key on the console to do the automatic reboot after a panic. Otherwise the debugging interface will wait forever for some remote debugger to attach in case of a panic.

ipkdbif_init(*kip*) In order to be able to find the debugging interface, the network driver must invoke **ipkdbif_init**() with *kip* specifying a *struct ipkdb_if* plus some additional parameters that allow it to access the devices registers, hopefully using bus_space(9) methods. In the *ipkdb_if* structure, the attach routine must initialize the following fields:

myenetaddr	fill this with the own ethernet address of the device/machine
flags	mark at least IPKDB_MYHW here
name	name of the device, only used for a message
start	routine called every time **ipkdb** is entered
leave	routine called every time **ipkdb** is left
receive	routine called to receive a packet
send	routine called to send a packet

Additional fields that may be set are:

myinetaddr	fill this with the own internet address, and mark IPKDB_MYIP in flags
port	may be used as a pointer to some device

ipkdbfbyte(*c*) This routine should fetch a byte from address *c*. It must not enter any trap handling code, but instead return −1 on inability to access the data.

ipkdbsbyte(*c*, *i*) This routine should set the byte pointed to by *c* to the value given as *i*. The routine must not enter any trap handling code. Furthermore it should reset the modification bit in the relevant page table entry to the value before the store.

SEE ALSO

ipkdb(4)

NAME

ISA, **isa_intr_alloc**, **isa_intr_establish**, **isa_intr_disestablish**, **isa_intr_evcnt**, **isa_dmamap_create**, **isa_dmamap_destroy**, **isa_dmamem_alloc**, **isa_dmamem_free**, **isa_dmamem_map**, **isa_dmamem_unmap**, **isa_malloc**, **isa_free**, **isa_dmastart**, **isa_dmaabort**, **isa_dmacount**, **isa_dmadone**, **isa_dmamaxsize**, **isa_drq_alloc**, **isa_drq_free**, **isa_drq_isfree**, **isa_dmacascade**, **isa_mappage** — Industry-standard Architecture

SYNOPSIS

```
#include <machine/bus.h>
#include <dev/isa/isareg.h>
#include <dev/isa/isavar.h>

int
isa_intr_alloc(isa_chipset_tag_t ic, int mask, int type, int *irq);

const struct evcnt *
isa_intr_evcnt(isa_chipset_tag_t ic, int irq);

void *
isa_intr_establish(isa_chipset_tag_t ic, int irq, int type, int level,
    int (*handler)(void *), void *arg);

void
isa_intr_disestablish(isa_chipset_tag_t ic, void *ih);

#include <dev/isa/isadmareg.h>
#include <dev/isa/isadmavar.h>

int
isa_dmamap_create(isa_chipset_tag_t ic, int chan, bus_size_t size,
    int flags);

void
isa_dmamap_destroy(isa_chipset_tag_t ic, int chan);

int
isa_dmamem_alloc(isa_chipset_tag_t ic, int chan, bus_size_t size,
    bus_addr_t *addrp, int flags);

void
isa_dmamem_free(isa_chipset_tag_t ic, int chan, bus_addr_t addr,
    bus_size_t size);

int
isa_dmamem_map(isa_chipset_tag_t ic, int chan, bus_addr_t addr,
    bus_size_t size, void **kvap, int flags);

void
isa_dmamem_unmap(isa_chipset_tag_t ic, int chan, void *kva, size_t size);

void *
isa_malloc(isa_chipset_tag_t ic, int chan, size_t size, int pool,
    int flags);

void
isa_free(void *addrp, int pool);
```

int
isa_dmastart(*isa_chipset_tag_t ic*, *int chan*, *bus_addr_t addr*, *bus_size_t size*, *struct lwp *lwp*, *int flags*, *int bf*);

void
isa_dmaabort(*isa_chipset_tag_t ic*, *int chan*);

bus_size_t
isa_dmacount(*isa_chipset_tag_t ic*, *int chan*);

void
isa_dmadone(*isa_chipset_tag_t ic*, *int chan*);

bus_size_t
isa_dmamaxsize(*isa_chipset_tag_t ic*, *int chan*);

int
isa_drq_alloc(*isa_chipset_tag_t ic*, *int chan*);

int
isa_drq_free(*isa_chipset_tag_t ic*, *int chan*);

int
isa_drq_isfree(*isa_chipset_tag_t ic*, *int chan*);

int
isa_dmacascade(*isa_chipset_tag_t ic*, *int chan*);

paddr_t
isa_mappage(*void *mem*, *off_t offset*, *int prot*);

DESCRIPTION

The machine-independent **ISA** subsystem provides support for the ISA bus.

The ISA bus was introduced on the IBM PC/AT. It is an extension to the original bus found on the original IBM PC. The ISA bus is essentially the host bus of the Intel 80286 processor, however the widespread acceptance of the bus as a de facto standard has seen it appear on systems without Intel processors.

The ISA bus has a 16-bit data bus, a 24-bit memory address bus, a 16-bit I/O address bus, and operates at 8MHz. It provides 15 interrupt lines and 8 DMA channels supporting DMA transfers of 64KB or 128KB transfers depending on the width of the channel being used. Historically, some devices only decoded the 10 lowest bits of the I/O address bus, preventing use of the full 16-bit address space.

On newer machines, the ISA bus is no longer connected directly to the host bus, and is usually connected via a PCI-ISA bridge. Either way, the bus looks the same to the device driver.

DATA TYPES

Drivers for devices attached to the **ISA** bus will make use of the following data types:

isa_chipset_tag_t
: Chipset tag for the ISA bus.

struct isa_attach_args
: Location hints for devices are recorded in this structure. It contains the following members:

```
bus_space_tag_t ia_iot;              /* isa i/o space tag */
bus_space_tag_t ia_memt;      /* isa mem space tag */
bus_dma_tag_t ia_dmat;        /* DMA tag */
isa_chipset_tag_t ia_ic;
```

```
int ia_iobase;                  /* base i/o address */
int ia_iosize;                  /* span of ports used */
int ia_maddr;                   /* physical mem addr */
u_int ia_msize;                         /* size of memory */
int ia_irq;                     /* interrupt request */
int ia_drq;                     /* DMA request */
int ia_drq2;                    /* second DMA request */
void *ia_aux;                   /* driver specific */
```

FUNCTIONS

isa_intr_alloc(*ic*, *mask*, *type*, *irq*)

This function is generally not required by device drivers. It is used by bridges attaching other busses to the ISA bus.

isa_intr_evcnt(*ic*, *irq*)

Returns the event counter associated with interrupt line *irq*.

isa_intr_establish(*ic*, *irq*, *type*, *level*, *handler*, *arg*)

To establish an ISA interrupt handler, a driver calls **isa_intr_establish**() with the interrupt number *irq*, type *type*, and level *level*. When the interrupt occurs the function *handler* is called with argument *arg*. Valid values for *type* are:

IST_NONE
: Reserve interrupt, but don't actually establish.

IST_EDGE
: Edge-triggered interrupt.

IST_LEVEL
: Level-triggered interrupt.

IST_PULSE
: Pulse-triggered interrupt.

isa_intr_establish() returns an opaque handle to an event descriptor if it succeeds, and returns NULL on failure.

isa_intr_disestablish(*ic*, *ih*)

Dis-establish the interrupt handler with handle *ih*. The handle was returned from **isa_intr_establish**().

isa_drq_alloc(*ic*, *chan*)

Reserves the DMA channel *chan* for future use. Normally, this call precedes an **isa_dmamap_create**() call. It is an error to start DMA on a channel that has not been reserved with **isa_drq_alloc**().

isa_drq_free(*ic*, *chan*)

Marks the DMA channel *chan* as available again.

isa_dmamap_create(*ic*, *chan*, *size*, *flags*)

Creates a DMA map for channel *chan*. It is initialised to accept maximum DMA transfers of size *size*. Valid values for the *flags* argument are the same as for **bus_dmamap_create**() (see bus_dma(9)). This function returns zero on success or an error value on failure.

isa_dmamap_destroy(*ic*, *chan*)

Destroy the DMA map for DMA channel *chan*.

isa_dmamem_alloc(*ic*, *chan*, *size*, *addrp*, *flags*)
Allocate DMA-safe memory of size *size* for channel *chan*. Valid values for the *flags* argument are the same as for **bus_dmamem_alloc**() (see bus_dma(9)). The bus-address of the memory is returned in *addrp*. This function returns zero on success or an error value on failure.

isa_dmamem_free(*ic*, *chan*, *addr*, *size*)
Frees memory previously allocated by **isa_dmamem_alloc**() for channel *chan*. The bus-address and size of the memory are specified by *addr* and *size* respectively.

isa_dmamem_map(*ic*, *chan*, *addr*, *size*, *kvap*, *flags*)
Maps DMA-safe memory (allocated with **isa_dmamem_alloc**()) specified by bus-address *addr* and of size *size* into kernel virtual address space for DMA channel *chan*. Valid values for the *flags* argument are the same as for **bus_dmamem_map**() (see bus_dma(9)). The kernel virtual address is returned in *kvap*. This function returns zero on success or an error value on failure.

isa_dmamem_unmap(*ic*, *chan*, *kva*, *size*)
Unmaps memory (previously mapped with **isa_dmamem_map**()) of size *size* for channel *chan*. The kernel virtual address space used by the mapping is freed.

isa_malloc(*ic*, *chan*, *size*, *pool*, *flags*)
This function is a shortcut for allocating and mapping DMA-safe memory in a single step. The arguments correspond with the arguments to **isa_dmamem_alloc**() and **isa_dmamem_map**(). The argument *pool* is a pool to record the memory allocation. This function returns a pointer to the DMA-safe memory.

isa_free(*addrp*, *pool*)
This function is a shortcut for unmapping and deallocating DMA-safe memory in a single step. It replaces **isa_dmamem_unmap**() and **isa_dmamem_free**(). The argument *addrp* is the pointer to the DMA-safe memory returned by **isa_malloc**(). The argument *pool* is the same as the value passed to **isa_malloc**().

isa_dmastart(*ic*, *chan*, *addr*, *size*, *lwp*, *flags*, *bf*)
Load DMA memory specified by address *addr* of size *size* into the DMA controller at channel *chan* and set it in motion. The argument *lwp* is used to indicate the address space in which the buffer is located. If NULL, the buffer is assumed to be in kernel space. Otherwise, the buffer is assumed to be in lwp *lwp 's* address space. The argument *flags* describes the type of ISA DMA. Valid values are:

DMAMODE_WRITE
DMA transfer from host to device.

DMAMODE_READ
DMA transfer to host from device.

DMAMODE_SINGLE
Transfer buffer once and stop.

DMAMODE_DEMAND
Demand mode.

DMAMODE_LOOP
Transfer buffer continuously in loop until notified to stop.

DMAMODE_LOOPDEMAND
Transfer buffer continuously in loop and demand mode.

The argument *bf* is the bus-space flags. Valid values are the same as for **bus_dmamap_load**()

(see bus_dma(9)).

isa_dmaabort(*ic*, *chan*)
Abort a DMA transfer on channel *chan*.

isa_dmacount(*ic*, *chan*)
Returns the offset in the DMA memory of the current DMA transfer on channel *chan*.

isa_dmadone(*ic*, *chan*)
Unloads the DMA memory on channel *chan* after a DMA transfer has completed.

isa_dmamaxsize(*ic*, *chan*)
Returns the maximum allowable DMA transfer size for channel *chan*.

isa_drq_isfree(*ic*, *chan*)
If the *ia_drq* or *ia_drq2* members of *struct isa_attach_args* are wildcarded, then the driver is expected to probe the hardware for valid DMA channels. In this case, the driver can check to see if the hardware-supported DMA channel *chan* is available for use.

isa_dmacascade(*ic*, *chan*)
Programs the 8237 DMA controller channel *chan* to accept external DMA control by the device hardware.

isa_mappage(*mem*, *offset*, *prot*)
Provides support for user mmap(2)'ing of DMA-safe memory.

AUTOCONFIGURATION

The ISA bus is an indirect-connection bus. During autoconfiguration each driver is required to probe the bus for the presence of a device. An ISA driver will receive a pointer to *struct isa_attach_args* hinting at "locations" on the ISA bus where the device may be located. They should use the *ia_iobase*, *ia_iosize*, *ia_maddr*, and *ia_msize* members. Not all of these hints will be necessary; locators may be wildcarded with IOBASEUNK and MADDRUNK for *ia_iobase* and *ia_maddr* respectively. If a driver can probe the device for configuration information at default locations, it may update the members of *struct isa_attach_args*. The IRQ and DMA locators can also be wildcarded with IRQUNK and DRQUNK respectively.

During the driver attach step, the I/O and memory address spaces should be mapped (see bus_space(9)).

DMA SUPPORT

Extensive DMA facilities are provided for the ISA bus. A driver can use up to two DMA channels simultaneously. The DMA channels allocated during autoconfiguration are passed to the driver during the driver attach using the *ia_drq* and *ia_drq2* members of *struct isa_attach_args*.

Before allocating resources for DMA transfers on the ISA bus, a driver should check the maximum allowable DMA transfer size for the DMA channel using **isa_dmamaxsize**().

A DMA map should be created first using **isa_dmamap_create**(). A DMA map describes how DMA memory is loaded into the DMA controllers. Only DMA-safe memory can be used for DMA transfers. DMA-safe memory is allocated using **isa_dmamem_alloc**(). The memory allocated by **isa_dmamem_alloc**() must now be mapped into kernel virtual address space by **isa_dmamem_map**() so that it can be accessed by the driver.

For a DMA transfer from the host to the device, the driver will fill the DMA memory with the data to be transferred. The DMA-transfer of the memory is started using **isa_dmastart**() with *flags* containing DMAMODE_WRITE. When the DMA transfer is completed, a call to **isa_dmadone**() cleans up the DMA transfer by unloading the memory from the controller.

For a DMA transfer from the device to the host, the DMA-transfer is started using **isa_dmastart**() with *flags* containing DMAMODE_READ. When the DMA transfer is completed, a call to **isa_dmadone**() cleans up the DMA transfer by unloading the memory from the controller. The memory can now be access by the driver.

When the DMA resources are no longer required they should be released using **isa_dmamem_unmap**(), **isa_dmamem_free**() and **isa_dmamap_destroy**().

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent ISA subsystem can be found. All pathnames are relative to `/usr/src`.

The ISA subsystem itself is implemented within the files `sys/dev/isa/isa.c` and `sys/dev/isa/isadma.c`.

SEE ALSO

`isa`(4), `autoconf`(9), `bus_dma`(9), `bus_space`(9), `driver`(9), `isapnp`(9)

HISTORY

The machine-independent **ISA** subsystem appeared in NetBSD 1.2.

BUGS

The previous behaviour of **isa_intr_establish**() was to invoke **panic**() on failure. **isa_intr_establish**() now returns NULL on failure. Some old drivers written for the former behaviour discard the return value.

NAME

ISAPNP, **isapnp_devmatch**, **isapnp_config**, **isapnp_unconfig** — Plug 'n' Play ISA bus

SYNOPSIS

```
#include <machine/bus.h>
#include <dev/isa/isareg.h>
#include <dev/isa/isavar.h>
#include <dev/isapnp/isapnpreg.h>
#include <dev/isapnp/isapnpvar.h>
#include <dev/isapnp/isapnpdevs.h>

int
isapnp_devmatch(const struct isapnp_attach_args *ipa,
    const struct isapnp_devinfo *dinfo, int *variant);

int
isapnp_config(bus_space_tag_t iot, bus_space_tag_t memt,
    struct isapnp_attach_args *ipa);

void
isapnp_unconfig(bus_space_tag_t iot, bus_space_tag_t memt,
    struct isapnp_attach_args *ipa);
```

DESCRIPTION

The machine-independent **ISAPNP** subsystem provides support for ISAPNP devices. ISAPNP devices were developed to support "plug and play" connection on the ISA bus. In all other aspects, the ISAPNP bus is same as the ISA bus (see isa(9)).

Devices on the ISAPNP bus are uniquely identified by a 7-character string. Resources, such as I/O address space and interrupts, should be allocated to the devices by the machine firmware. On some machine the firmware seems doesn't work correctly and NetBSD will attempt to allocate resources as necessary.

DATA TYPES

Drivers attached to the ISAPNP bus will make use of the following data types:

struct isapnp_matchinfo

NetBSD kernel contains a database of known ISAPNP devices. Each entry in the database has a *struct isapnp_matchinfo*. It contains the following members:

```
const char *name;              /* device id string */
int variant;                   /* variant flag */
```

struct isapnp_devinfo

Defines the devices supported by a driver. It contains pointer to an array of supported *struct isapnp_matchinfo* structures and a pointer to another array of compatibility devices. It contains the following members:

```
struct isapnp_matchinfo *devlogic;
int nlogic;
struct isapnp_matchinfo *devcompat;
int ncompat;
```

struct isapnp_region

Describes ISAPNP bus-space regions. It contains the following members:

```
bus_space_handle_t h;
uint32_t base;
uint32_t length;
```

struct isapnp_pin

Describes the wiring of interrupts and DMA pins from the ISAPNP bus onto the host processor. It contains the following members:

```
uint8_t  num;
uint8_t  flags:4;
uint8_t  type:4;
uint16_t bits;
```

struct isapnp_attach_args

A structure used to inform the driver of the device properties. It contains the following members:

```
bus_space_tag_t       ipa_iot;       /* isa i/o space tag */
bus_space_tag_t       ipa_memt;      /* isa mem space tag */
bus_dma_tag_t  ipa_dmat;      /* isa dma tag */
isa_chipset_tag_t     ipa_ic;
struct isapnp_region    ipa_io[ISAPNP_NUM_IO];
struct isapnp_region    ipa_mem[ISAPNP_NUM_MEM];
struct isapnp_region    ipa_mem32[ISAPNP_NUM_MEM32];
struct isapnp_pin       ipa_irq[ISAPNP_NUM_IRQ];
struct isapnp_pin       ipa_drq[ISAPNP_NUM_DRQ];
```

FUNCTIONS

isapnp_devmatch(*ipa*, *dinfo*, *variant*)

Matches the device described by the attachment *ipa* with the device-match information in *dinfo*. If the device is matched, **isapnp_devmatch**() returns a non-zero value and variant is the flag describing the device variant. **isapnp_devmatch**() returns zero if the device is not found.

isapnp_config(*iot*, *memt*, *ipa*)

Allocate device resources specified by *ipa*. The device is mapped into the I/O and memory bus spaces specified by bus-space tags *iot* and *memt* respectively. The *ipa_io*, *ipa_mem*, *ipa_mem32*, *ipa_irq*, and *ipa_drq* members of *ipa* are updated to reflect the allocated and mapped resources. **isapnp_config**() returns zero on success and non-zero on error.

isapnp_unconfig(*iot*, *memt*, *ipa*)

Free the resources allocated by **isapnp_config**().

AUTOCONFIGURATION

During autoconfiguration, an ISAPNP driver will receive a pointer to *struct isapnp_attach_args* describing the device attached to the ISAPNP bus. Drivers match the device using **ispnp_devmatch**().

During the driver attach step, driver should initially allocate and map resources using **isapnp_config**(). The I/O (memory) bus-space resources can be accessed using the bus-space tag *ipa_iot* (*ipa_memt*) and the bus-space handle *ipa_io[0].h* (*ipa_mem[0].h*) members of *ipa*.

Interrupts should be established using **isa_intr_establish**() (see isa(9)) with the IRQ specified by the *ipa_irq[0].num* member of *ipa*. Similarly, the standard isa(9) DMA interface should be used with the *ipa_drq[0].num* member of *ipa*.

DMA SUPPORT

Extensive DMA facilities are provided through the `isa`(9) DMA facilities.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent **`ISAPNP`** subsystem can be found. All pathnames are relative to `/usr/src`.

The **`ISAPNP`** subsystem itself is implemented within the file `sys/dev/isapnp/isapnp.c`. The database of the known devices exists within the file `sys/dev/isapnp/isapnpdevs.c` and is generated automatically from the file `sys/dev/isapnp/isapnpdevs`. New devices should be added to this file. The database can be regenerated using the Makefile `sys/dev/isapnp/Makefile.isapnpdevs`.

SEE ALSO

`isa`(4), `isapnp`(4), `pnpbios`(4), `autoconf`(9), `bus_dma`(9), `bus_space`(9), `driver`(9), `isa`(9)

Plug and Play ISA Specification V1.0a, May 5 1994.

HISTORY

The machine-independent ISAPNP subsystem appear in NetBSD 1.3.

NAME

isr_add, **isr_add_autovect**, **isr_add_vectored**, **isr_add_custom** — establish interrupt handler

SYNOPSIS

```
#include <sun3/autoconf.h>

typedef int (*isr_func_t)(void *);

void
isr_add_autovect(isr_func_t fun, void *arg, int level);

void
isr_add_vectored(isr_func_t fun, void *arg, int pri, int vec);

void
isr_add_custom(int level, void *fun);
```

DESCRIPTION

The **isr_add** functions establish interrupt handlers into the system interrupt dispatch table and are typically called from device drivers during the autoconfiguration process.

There are two types of interrupts in the Motorola 68000 architecture, which differ in the way that an interrupt request is mapped to a dispatch function within the interrupt vector table.

When the CPU detects an asserted signal on one of its interrupt request lines, it suspends normal instruction execution and begins an interrupt acknowledge cycle on the system bus. During this cycle the interrupting device directs how the CPU is to dispatch its interrupt request.

If the interrupting device is integrated tightly with the system bus, it provides an 8-bit interrupt vector number to the CPU and a **vectored** interrupt occurs. This vector number points to a vector entry within the interrupt vector table to which instruction execution is immediately transfered.

If the interrupting device cannot provide a vector number, it asserts a specialized bus line and an **autovectored** interrupt occurs. The vector number to use is determined by adding the interrupt priority (0–6) to an autovector base (typically 18 hexadecimal).

isr_add_autovect()
: Adds the function *fun* to the list of interrupt handlers to be called during an autovectored interrupt of priority *level*. The pointer *arg* is passed to the function as its first argument.

isr_add_vectored()
: Adds the function *fun* to the list of interrupt handlers to be called during a vectored interrupts of priority *pri* at dispatch vector number *vec*. The pointer *arg* is passed to the function as its first argument.

isr_add_custom()
: Establish function *fun* as the interrupt handler for vector *level*. The autovector base number is automatically added to *level*.

  fun is called directly as the dispatch handler and must handle all of the specifics of saving the processor state and returning from a processor exception. These requirements generally dictate that *fun* be written in assembler.

CODE REFERENCES

`sys/arch/sun3/sun3/isr.c`

REFERENCES

MC68030 User's Manual, Third edition, MC68030UM/AD Rev 2, Motorola Inc.

BUGS

There is no way to remove a handler once it has been added.

NAME

itimerfix — check that a timeval value is valid, and correct

SYNOPSIS

```
#include <sys/time.h>

int
itimerfix(struct timeval *tv);
```

DESCRIPTION

The **itimerfix** function checks that the value in *tv* is valid (0 ≤ tv->tv_sec && 0 ≤ tv->tv_usec < 1000000), and that the total time represented is at least one *tick*, or zero.

If the total represented time is nonzero and smaller than tick, it is adjusted to exactly one tick.

RETURN VALUES

itimerfix returns 0 on success or EINVAL if *tv* is invalid.

SEE ALSO

nanosleep(2), poll(2), select(2), setitimer(2)

NAME

kauth — kernel authorization framework

SYNOPSIS

```
#include <sys/kauth.h>
```

DESCRIPTION

kauth, or kernel authorization, is the subsystem managing all authorization requests inside the kernel. It manages user credentials and rights, and can be used to implement a system-wide security policy. It allows external modules to plug-in the authorization process.

kauth introduces some new concepts, namely "scopes" and "listeners", which will be detailed together with other useful information for kernel developers in this document.

Types

Some **kauth** types include the following:

kauth_cred_t Representing credentials that can be associated with an object. Includes user- and group-ids (real, effective, and save) as well as group membership information.

kauth_scope_t Describes a scope.

kauth_listener_t
Describes a listener.

Terminology

kauth operates in various "scopes", each scope holding a group of "listeners".

Each listener works as a callback for when an authorization request within the scope is made. When such a request is made, all listeners on the scope are passed common information such as the credentials of the request context, an identifier for the requested operation, and possibly other information as well.

Every listener examines the passed information and returns its decision regarding the requested operation. It can either allow, deny, or defer the operation -- in which case, the decision is left to the other listeners.

For an operation to be allowed, all listeners must not return any deny or defer decisions.

Scopes manage listeners that operate in the same aspect of the system.

Kernel Programming Interface

kauth exports a KPI that allows developers both of NetBSD and third-party products to authorize requests, access and modify credentials, create and remove scopes and listeners, and perform other miscellaneous operations on credentials.

Authorization Requests

kauth provides a single authorization request routine, which all authorization requests go through. This routine dispatches the request to the listeners of the appropriate scope, together with four optional user-data variables, and returns the augmented result.

It is declared as

int **kauth_authorize_action**(*kauth_scope_t scope*, *kauth_cred_t cred*, *kauth_action_t op*, *void *arg0*, *void *arg1*, *void *arg2*, *void *arg3*)

An authorization request can return one of two possible values. Zero indicates success -- the operation is allowed; EPERM (see errno(2)) indicates failure -- the operation is denied.

Each scope has its own authorization wrapper, to make it easy to call from various places by eliminating the need to specify the scope and/or cast values. The authorization wrappers are detailed in each scope's section.

kauth_authorize_action() has several special cases, when it will always allow the request. These are for when the request is issued by the kernel itself (indicated by the credentials being either NOCRED or FSCRED), or when there was no definitive decision from any of the listeners (i.e., it was not explicitly allowed or denied) and no security model was loaded.

Generic Scope

The generic scope, "org.netbsd.kauth.generic", manages generic authorization requests in the kernel.

The authorization wrapper for this scope is declared as

int **kauth_authorize_generic**(*kauth_cred_t cred*, *kauth_action_t op*, *void *arg0*)

The following operations are available for this scope:

KAUTH_GENERIC_ISSUSER
: Checks whether the credentials belong to the super-user.

  Using this request is strongly discouraged and should only be done as a temporary placeholder, as it is breaking the separation between the interface for authorization requests from the back-end implementation.

KAUTH_GENERIC_CANSEE
: Checks whether an object with one set of credentials can access information about another object, possibly with a different set of credentials.

  arg0 contains the credentials of the object looked at.

  This request should be issued only in cases where generic credentials check is required; otherwise it is recommended to use the object-specific routines.

System Scope

The system scope, "org.netbsd.kauth.system", manages authorization requests affecting the entire system.

The authorization wrapper for this scope is declared as

int **kauth_authorize_system**(*kauth_cred_t cred*, *kauth_action_t op*, *enum kauth_system_req req*, *void *arg1*, *void *arg2*, *void *arg3*)

The following requests are available for this scope:

KAUTH_SYSTEM_ACCOUNTING
: Check if enabling/disabling accounting allowed.

KAUTH_SYSTEM_CHROOT
: *req* can be any of the following:

  KAUTH_REQ_SYSTEM_CHROOT_CHROOT
  : Check if calling chroot(2) is allowed.

  KAUTH_REQ_SYSTEM_CHROOT_FCHROOT
  : Check if calling fchroot(2) is allowed.

KAUTH_SYSTEM_CPU
: Check CPU-manipulation access.

req can be any of the following:

KAUTH_REQ_SYSTEM_CPU_SETSTATE
: Set CPU state, including setting it online or offline.

KAUTH_SYSTEM_DEBUG
: This request concentrates several debugging-related operations. *req* can be any of the following:

  KAUTH_REQ_SYSTEM_DEBUG_IPKDB
  : Check if using ipkdb(4) is allowed.

KAUTH_SYSTEM_FILEHANDLE
: Check if filehandle operations allowed.

KAUTH_SYSTEM_LKM
: Check if an LKM request is allowed.

  arg1 is the command.

KAUTH_SYSTEM_MKNOD
: Check if creating devices is allowed.

KAUTH_SYSTEM_MOUNT
: Check if mount-related operations are allowed.

  req can be any of the following:

  KAUTH_REQ_SYSTEM_MOUNT_GET
  : Check if retrieving information about a mount is allowed. *arg1* is a *struct mount* * with the mount structure in question, *arg2* is a *void* * with file-system specific data, if any.

  KAUTH_REQ_SYSTEM_MOUNT_NEW
  : Check if mounting a new file-system is allowed.

    arg1 is the *struct vnode* * on which the file-system is to be mounted, *arg2* is an *int* with the mount flags, and *arg3* is a *void* * with file-system specific data, if any.

  KAUTH_REQ_SYSTEM_MOUNT_UNMOUNT
  : Checks if unmounting a file-system is allowed.

    arg1 is a *struct mount* * with the mount in question.

  KAUTH_REQ_SYSTEM_MOUNT_UPDATE
  : Checks if updating an existing mount is allowed.

    arg1 is the *struct mount* * of the existing mount, *arg2* is an *int* with the new mount flags, and *arg3* is a *void* * with file-system specific data, if any.

KAUTH_SYSTEM_PSET
: Check processor-set manipulation.

  req can be any of the following:

  KAUTH_REQ_SYSTEM_PSET_ASSIGN
  : Change processor-set processor assignment.

KAUTH_REQ_SYSTEM_PSET_BIND
Bind an LWP to a processor-set.

KAUTH_REQ_SYSTEM_PSET_CREATE
Create a processor-set.

KAUTH_REQ_SYSTEM_PSET_DESTROY
Destroy a processor-set.

KAUTH_SYSTEM_REBOOT
Check if rebooting is allowed.

KAUTH_SYSTEM_SETIDCORE
Check if changing coredump settings for set-id processes is allowed.

KAUTH_SYSTEM_SWAPCTL
Check if privileged swapctl(2) requests are allowed.

KAUTH_SYSTEM_SYSCTL
This requests operations related to sysctl(9). *req* indicates the specific request and can be one of the following:

KAUTH_REQ_SYSTEM_SYSCTL_ADD
Check if adding a sysctl(9) node is allowed.

KAUTH_REQ_SYSTEM_SYSCTL_DELETE
Check if deleting a sysctl(9) node is allowed.

KAUTH_REQ_SYSTEM_SYSCTL_DESC
Check if adding description to a sysctl(9) node is allowed.

KAUTH_REQ_SYSTEM_SYSCTL_PRVT
Check if accessing private sysctl(9) nodes is allowed.

KAUTH_SYSTEM_TIME
This request groups time-related operations. *req* can be any of the following:

KAUTH_REQ_SYSTEM_TIME_ADJTIME
Check if changing the time using adjtime(2) is allowed.

KAUTH_REQ_SYSTEM_TIME_NTPADJTIME
Check if setting the time using ntp_adjtime(2) is allowed.

KAUTH_REQ_SYSTEM_TIME_SYSTEM
Check if changing the time (usually via settimeofday(2)) is allowed.

arg1 is a *struct timespec* * with the new time, *arg2* is a *struct timeval* * with the delta from the current time, *arg3* is a *bool* indicating whether the caller is a device context (eg. /dev/clockctl) or not.

KAUTH_REQ_SYSTEM_TIME_RTCOFFSET
Check if changing the RTC offset is allowed.

KAUTH_REQ_SYSTEM_TIME_TIMECOUNTERS
Check if manipulating timecounters is allowed.

Process Scope

The process scope, "org.netbsd.kauth.process", manages authorization requests related to processes in the system.

The authorization wrapper for this scope is declared as

int **kauth_authorize_process**(*kauth_cred_t cred*, *kauth_action_t op*, *struct proc *p*, *void *arg1*, *void *arg2*, *void *arg3*)

The following operations are available for this scope:

KAUTH_PROCESS_KTRACE
: Checks whether an object with one set of credentials can ktrace(1) another process *p*, possibly with a different set of credentials.

  If *arg1* is KAUTH_REQ_PROCESS_KTRACE_PERSISTENT, this checks if persistent tracing can be done. Persistent tracing maintains the trace across a set-user-id/set-group-id exec(2), and normally requires privileged credentials.

KAUTH_PROCESS_PROCFS
: Checks whether object with passed credentials can use *procfs* to access process *p*.

  arg1 is the *struct pfsnode* * for the target element in the target process, and *arg2* is the access type, which can be either KAUTH_REQ_PROCESS_PROCFS_CTL, KAUTH_REQ_PROCESS_PROCFS_READ, KAUTH_REQ_PROCESS_PROCFS_RW, or KAUTH_REQ_PROCESS_PROCFS_WRITE, indicating *control*, *read*, *read-write*, or *write* access respectively.

KAUTH_PROCESS_PTRACE
: Checks whether object with passed credentials can use ptrace(2) to access process *p*.

  arg1 is the ptrace(2) command.

KAUTH_PROCESS_CANSEE
: Checks whether an object with one set of credentials can access information about another process, possibly with a different set of credentials.

  arg1 indicates the class of information being viewed, and can either of KAUTH_REQ_PROCESS_CANSEE_ARGS, KAUTH_REQ_PROCESS_CANSEE_ENTRY, KAUTH_REQ_PROCESS_CANSEE_ENV, or KAUTH_REQ_PROCESS_CANSEE_OPENFILES.

KAUTH_PROCESS_SCHEDULER_GETAFFINITY
: Checks whether viewing the scheduler affinity is allowed.

KAUTH_PROCESS_SCHEDULER_SETAFFINITY
: Checks whether setting the scheduler affinity is allowed.

KAUTH_PROCESS_SCHEDULER_GETPARAMS
: Checks whether viewing the scheduler policy and parameters is allowed.

KAUTH_PROCESS_SCHEDULER_SETPARAMS
: Checks whether modifying the scheduler policy and parameters is allowed.

KAUTH_PROCESS_SIGNAL
: Checks whether an object with one set of credentials can post signals to another process.

  p is the process the signal is being posted to, and *arg1* is the signal number.

KAUTH_PROCESS_CORENAME
: Controls access to process corename.

  arg1 can be KAUTH_REQ_PROCESS_CORENAME_GET or KAUTH_REQ_PROCESS_CORENAME_SET, indicating access to read or write the process' corename, respectively.

When modifying the corename, *arg2* holds the new corename to be used.

KAUTH_PROCESS_FORK
: Checks if the process can fork. *arg1* is an *int* indicating how many processes exist on the system at the time of the check.

KAUTH_PROCESS_KEVENT_FILTER
: Checks whether setting a process kevent(2) filter is allowed.

KAUTH_PROCESS_NICE
: Checks whether the *nice* value of *p* can be changed to *arg1*.

KAUTH_PROCESS_RLIMIT
: Controls access to process resource limits.

  arg1 can be KAUTH_REQ_PROCESS_RLIMIT_GET or KAUTH_REQ_PROCESS_RLIMIT_SET, indicating access to read or write the process' resource limits, respectively.

  When modifying resource limits, *arg2* is the new value to be used and *arg3* indicates which resource limit is to be modified.

KAUTH_PROCESS_SETID
: Check if changing the user- or group-ids, groups, or login-name for *p* is allowed.

KAUTH_PROCESS_STOPFLAG
: Check if setting the stop flags for exec(3), exit(3), and fork(2) is allowed.

  arg1 indicates the flag, and can be either P_STOPEXEC, P_STOPEXIT, or P_STOPFORK respectively.

Network Scope

The network scope, "org.netbsd.kauth.network", manages networking-related authorization requests in the kernel.

The authorization wrapper for this scope is declared as

int **kauth_authorize_network**(*kauth_cred_t cred*, *kauth_action_t op*, *enum kauth_network_req req*, *void *arg1*, *void *arg2*, *void *arg3*)

The following operations are available for this scope:

KAUTH_NETWORK_ALTQ
: Checks if an ALTQ operation is allowed.

  req indicates the ALTQ subsystem in question, and can be one of the following:

  KAUTH_REQ_NETWORK_ALTQ_AFMAP
  KAUTH_REQ_NETWORK_ALTQ_BLUE
  KAUTH_REQ_NETWORK_ALTQ_CBQ
  KAUTH_REQ_NETWORK_ALTQ_CDNR
  KAUTH_REQ_NETWORK_ALTQ_CONF
  KAUTH_REQ_NETWORK_ALTQ_FIFOQ
  KAUTH_REQ_NETWORK_ALTQ_HFSC
  KAUTH_REQ_NETWORK_ALTQ_JOBS
  KAUTH_REQ_NETWORK_ALTQ_PRIQ
  KAUTH_REQ_NETWORK_ALTQ_RED

KAUTH_REQ_NETWORK_ALTQ_RIO
KAUTH_REQ_NETWORK_ALTQ_WFQ

KAUTH_NETWORK_BIND
Checks if a bind(2) request is allowed.

req allows to indicate the type of the request to structure listeners and callers easier. Supported request types:

KAUTH_REQ_NETWORK_BIND_PRIVPORT
Checks if binding to a privileged/reserved port is allowed.

KAUTH_NETWORK_FIREWALL
Checks if firewall-related operations are allowed.

req indicates the sub-action, and can be one of the following:

KAUTH_REQ_NETWORK_FIREWALL_FW
Modification of packet filtering rules.

KAUTH_REQ_NETWORK_FIREWALL_NAT
Modification of NAT rules.

KAUTH_NETWORK_INTERFACE
Checks if network interface-related operations are allowed.

arg1 is (optionally) the *struct ifnet* * associated with the interface. *arg2* is (optionally) an *int* describing the interface-specific operation. *arg3* is (optionally) a pointer to the interface-specific request structure. *req* indicates the sub-action, and can be one of the following:

KAUTH_REQ_NETWORK_INTERFACE_GET
Check if retrieving information from the device is allowed.

KAUTH_REQ_NETWORK_INTERFACE_GETPRIV
Check if retrieving privileged information from the device is allowed.

KAUTH_REQ_NETWORK_INTERFACE_SET
Check if setting parameters on the device is allowed.

KAUTH_REQ_NETWORK_INTERFACE_SETPRIV
Check if setting privileged parameters on the device is allowed.

Note that unless the *struct ifnet* * for the interface was passed in *arg1*, there's no way to tell what structure *arg3* is.

KAUTH_NETWORK_FORWSRCRT
Checks whether status of forwarding of source-routed packets can be modified or not.

KAUTH_NETWORK_NFS
Check is an NFS related operation is allowed.

req can be any of the following:

KAUTH_REQ_NETWORK_NFS_EXPORT
Check if modifying the NFS export table is allowed.

KAUTH_REQ_NETWORK_NFS_SVC
Check if access to the NFS nfssvc(2) syscall is allowed.

KAUTH_NETWORK_ROUTE
: Checks if a routing-related request is allowed.

  arg1 is the *struct rt_msghdr* * for the request.

KAUTH_NETWORK_SOCKET
: Checks if a socket related operation is allowed.

  req allows to indicate the type of the request to structure listeners and callers easier. Supported request types:

  KAUTH_REQ_NETWORK_SOCKET_RAWSOCK
  : Checks if opening a raw socket is allowed.

  KAUTH_REQ_NETWORK_SOCKET_OPEN
  : Checks if opening a socket is allowed. *arg1*, *arg2*, and *arg3* are all *int* parameters describing the domain, socket type, and protocol, respectively.

  KAUTH_REQ_NETWORK_SOCKET_CANSEE
  : Checks if looking at the socket passed is allowed.

    arg1 is a *struct socket* * describing the socket.

Machine-dependent Scope

The machine-dependent (machdep) scope, "org.netbsd.kauth.machdep", manages machine-dependent authorization requests in the kernel.

The authorization wrapper for this scope is declared as

int **kauth_authorize_machdep**(*kauth_cred_t cred*, *kauth_action_t op*, *void *arg0*, *void *arg1*, *void *arg2*, *void *arg3*)

The actions on this scope provide a set that may or may not affect all platforms. Below is a list of available actions, along with which platforms are affected by each.

KAUTH_MACHDEP_IOPERM_GET
: Request to get the I/O permission level. Affects *amd64*, *i386*, *xen*.

KAUTH_MACHDEP_IOPERM_SET
: Request to set the I/O permission level. Affects *amd64*, *i386*, *xen*.

KAUTH_MACHDEP_IOPL
: Request to set the I/O privilege level. Affects *amd64*, *i386*, *xen*.

KAUTH_MACHDEP_LDT_GET
: Request to get the LDT (local descriptor table). Affects *amd64*, *i386*, *xen*.

KAUTH_MACHDEP_LDT_SET
: Request to set the LDT (local descriptor table). Affects *amd64*, *i386*, *xen*.

KAUTH_MACHDEP_MTRR_GET
: Request to get the MTRR (memory type range registers). Affects *amd64*, *i386*, *xen*.

KAUTH_MACHDEP_MTRR_SET
: Request to set the MTRR (memory type range registers). Affects *amd64*, *i386*, *xen*.

KAUTH_MACHDEP_UNMANAGEDMEM
: Request to access unmanaged memory. Affects *alpha*, *amd64*, *arm*, *i386*, *powerpc*, *sh3*, *vax*, *xen*.

Device Scope

The device scope, "org.netbsd.kauth.device", manages authorization requests related to devices on the system. Devices can be, for example, terminals, tape drives, and any other hardware. Network devices specifically are handled by the *network* scope.

In addition to the standard authorization wrapper:

int **kauth_authorize_device**(*kauth_cred_t cred*, *kauth_action_t op*, *void *arg0*, *void *arg1*, *void *arg2*, *void *arg3*)

this scope provides authorization wrappers for various device types.

int **kauth_authorize_device_tty**(*kauth_cred_t cred*, *kauth_action_t op*, *struct tty *tty*)

Authorizes requests for *terminal devices* on the system. The third argument, *tty*, is the terminal device in question. It is passed to the listener as *arg0*. The second argument, *op*, is the action and can be one of the following:

KAUTH_DEVICE_TTY_OPEN
: Open the terminal device pointed to by *tty*.

KAUTH_DEVICE_TTY_PRIVSET
: Set privileged settings on the terminal device pointed to by *tty*.

KAUTH_DEVICE_TTY_STI
: Use the "TIOCSTI" device ioctl(2), allowing to inject characters into the terminal buffer, simulating terminal input.

int **kauth_authorize_device_spec**(*kauth_cred_t cred*, *enum kauth_device_req req*, *struct vnode *vp*)

Authorizes requests for *special files*, usually disk devices, but also direct memory access, on the system.

It passes KAUTH_DEVICE_RAWIO_SPEC as the action to the listener, and accepts two arguments. *req*, passed to the listener as *arg0*, is access requested, and can be one of KAUTH_REQ_DEVICE_RAWIO_SPEC_READ, KAUTH_REQ_DEVICE_RAWIO_SPEC_WRITE, or KAUTH_REQ_DEVICE_RAWIO_SPEC_RW, representing read, write, or both read/write access respectively. *vp* is the vnode of the special file in question, and is passed to the listener as *arg1*.

Keep in mind that it is the responsibility of the security model developer to check whether the underlying device is a disk or the system memory, using **iskmemdev**():

```
if ((vp->v_type == VCHR) &&
    iskmemdev(vp->v_un.vu_specinfo->si_rdev))
        /* system memory access */
```

int **kauth_authorize_device_passthru**(*kauth_cred_t cred*, *dev_t dev*, *u_long mode*, *void *data*)

Authorizes hardware *passthru* requests, or user commands passed directly to the hardware. These have the potential of resulting in direct disk and/or memory access.

It passes KAUTH_DEVICE_RAWIO_PASSTHRU as the action to the listener, and accepts three arguments. *dev*, passed as *arg1* to the listener, is the device for which the request is made. *mode*, passed as *arg0* to the listener, is a generic representation of the access mode requested. It can be one or more (binary-OR'd) of the following:

KAUTH_REQ_DEVICE_RAWIO_PASSTHRU_READ
KAUTH_REQ_DEVICE_RAWIO_PASSTHRU_READCONF
KAUTH_REQ_DEVICE_RAWIO_PASSTHRU_WRITE
KAUTH_REQ_DEVICE_RAWIO_PASSTHRU_WRITECONF

data, passed as *arg2* to the listener, is device-specific data that may be associated with the request.

Credentials Scope

The credentials scope, "org.netbsd.kauth.cred", is a special scope used internally by the **kauth** framework to provide hooking to credential-related operations.

It is a "notify-only" scope, allowing hooking operations such as initialization of new credentials, credential inheritance during a fork, and copying and freeing of credentials. The main purpose for this scope is to give a security model a way to control the aforementioned operations, especially in cases where the credentials hold security model-private data.

Notifications are made using the following function, which is internal to **kauth**:

int **kauth_cred_hook**(*kauth_cred_t cred*, *kauth_action_t action*, *void *arg0*, *void *arg1*)

With the following actions:

KAUTH_CRED_COPY
: The credentials are being copied. *cred* are the credentials of the lwp context doing the copy, and *arg0* and *arg1* are both *kauth_cred_t* representing the "from" and "to" credentials, respectively.

KAUTH_CRED_FORK
: The credentials are being inherited from a parent to a child process during a fork.

  cred are the credentials of the lwp context doing the fork, and *arg0* and *arg1* are both *struct proc* * of the parent and child processes, respectively.

KAUTH_CRED_FREE
: The credentials in *cred* are being freed.

KAUTH_CRED_INIT
: The credentials in *cred* are being initialized.

Since this is a notify-only scope, all listeners are required to return KAUTH_RESULT_ALLOW.

Credentials Accessors and Mutators

kauth has a variety of accessor and mutator routines to handle *kauth_cred_t* objects.

The following routines can be used to access and modify the user- and group-ids in a *kauth_cred_t*:

uid_t **kauth_cred_getuid**(*kauth_cred_t cred*)
: Returns the real user-id from *cred*.

uid_t **kauth_cred_geteuid**(*kauth_cred_t cred*)
: Returns the effective user-id from *cred*.

uid_t **kauth_cred_getsvuid**(*kauth_cred_t cred*)
: Returns the saved user-id from *cred*.

void **kauth_cred_setuid**(*kauth_cred_t cred*, *uid_t uid*)
: Sets the real user-id in *cred* to *uid*.

void **kauth_cred_seteuid**(*kauth_cred_t cred*, *uid_t uid*)
Sets the effective user-id in *cred* to *uid*.

void **kauth_cred_setsvuid**(*kauth_cred_t cred*, *uid_t uid*)
Sets the saved user-id in *cred* to *uid*.

gid_t **kauth_cred_getgid**(*kauth_cred_t cred*)
Returns the real group-id from *cred*.

gid_t **kauth_cred_getegid**(*kauth_cred_t cred*)
Returns the effective group-id from *cred*.

gid_t **kauth_cred_getsvgid**(*kauth_cred_t cred*)
Returns the saved group-id from *cred*.

void **kauth_cred_setgid**(*kauth_cred_t cred*, *gid_t gid*)
Sets the real group-id in *cred* to *gid*.

void **kauth_cred_setegid**(*kauth_cred_t cred*, *gid_t gid*)
Sets the effective group-id in *cred* to *gid*.

void **kauth_cred_setsvgid**(*kauth_cred_t cred*, *gid_t gid*)
Sets the saved group-id in *cred* to *gid*.

u_int **kauth_cred_getrefcnt**(*kauth_cred_t cred*)
Return the reference count for *cred*.

The following routines can be used to access and modify the group list in a *kauth_cred_t*:

int **kauth_cred_ismember_gid**(*kauth_cred_t cred*, *gid_t gid*, *int *resultp*)
Checks if the group-id *gid* is a member in the group list of *cred*.

If it is, *resultp* will be set to one, otherwise, to zero.

The return value is an error code, or zero for success.

u_int **kauth_cred_ngroups**(*kauth_cred_t cred*)
Return the number of groups in the group list of *cred*.

gid_t **kauth_cred_group**(*kauth_cred_t cred*, *u_int idx*)
Return the group-id of the group at index *idx* in the group list of *cred*.

int **kauth_cred_setgroups**(*kauth_cred_t cred*, *gid_t *groups*, *size_t ngroups*, *uid_t gmuid*, *enum uio_seg seg*)
Copy *ngroups* groups from array pointed to by *groups* to the group list in *cred*, adjusting the number of groups in *cred* appropriately. *seg* should be either UIO_USERSPACE or UIO_SYSSPACE indicating whether *groups* is a user or kernel space address.

Any groups remaining will be set to an invalid value.

gmuid is unused for now, and to maintain interface compatibility with the Darwin KPI.

The return value is an error code, or zero for success.

int **kauth_cred_getgroups**(*kauth_cred_t cred*, *gid_t *groups*, *size_t ngroups*, *enum uio_seg seg*)
Copy *ngroups* groups from the group list in *cred* to the buffer pointed to by *groups*. *seg* should be either UIO_USERSPACE or UIO_SYSSPACE indicating whether *groups* is a user or kernel space address.

The return value is an error code, or zero for success.

Credential Private Data

kauth provides an interface to allow attaching security-model private data to credentials.

The use of this interface has two parts that can be divided to direct and indirect control of the private-data. Directly controlling the private data is done by using the below routines, while the indirect control is often dictated by events such as process fork, and is handled by listening on the credentials scope (see above).

Attaching private data to credentials works by registering a key to serve as a unique identifier, distinguishing various sets of private data that may be associated with the credentials. Registering, and deregistering, a key is done by using these routines:

int **kauth_register_key**(*const char *name*, *kauth_key_t *keyp*)
Register new key for private data for *name* (usually, the security model name). *keyp* will be used to return the key to be used in further calls.

The function returns 0 on success and an error code (see errno(2)) on failure.

int **kauth_deregister_key**(*kauth_key_t key*)
Deregister private data key *key*.

Once registered, private data may be manipulated by the following routines:

void **kauth_cred_setdata**(*kauth_cred_t cred*, *kauth_key_t key*, *void *data*)
Set private data for *key* in *cred* to be *data*.

void * **kauth_cred_getdata**(*kauth_cred_t cred*, *kauth_key_t key*)
Retrieve private data for *key* in *cred*.

Note that it is required to use the above routines every time the private data is changed, i.e., using **kauth_cred_getdata**() and later modifying the private data should be accompanied by a call to **kauth_cred_setdata**() with the "new" private data.

Credential Inheritance and Reference Counting

kauth provides an interface for handling shared credentials.

When a *kauth_cred_t* is first allocated, its reference count is set to 1. However, with time, its reference count can grow as more objects (processes, LWPs, files, etc.) reference it.

The following routines are available for managing credentials reference counting:

void **kauth_cred_hold**(*kauth_cred_t cred*)
Increases reference count to *cred* by one.

void **kauth_cred_free**(*kauth_cred_t cred*)
Decreases the reference count to *cred* by one.

If the reference count dropped to zero, the memory used by *cred* will be freed.

Credential inheritance happens during a fork(2), and is handled by the following function:

void **kauth_proc_fork**(*struct proc *parent*, *struct proc *child*)

When called, it references the parent's credentials from the child, and calls the credentials scope's hook with the KAUTH_CRED_FORK action to allow security model-specific handling of the inheritance to take place.

Credentials Memory Management

Data-structures for credentials, listeners, and scopes are allocated from memory pools managed by the pool(9) subsystem.

The *kauth_cred_t* objects have their own memory management routines:

kauth_cred_t **kauth_cred_alloc**(*void*)
Allocates a new *kauth_cred_t*, initializes its lock, and sets its reference count to one.

Conversion Routines

Sometimes it might be necessary to convert a *kauth_cred_t* to userland's view of credentials, a *struct uucred*, or vice versa.

The following routines are available for these cases:

void **kauth_uucred_to_cred**(*kauth_cred_t cred*, *const struct uucred *uucred*)
Convert userland's view of credentials to a *kauth_cred_t*.

This includes effective user- and group-ids, a number of groups, and a group list. The reference count is set to one.

Note that **kauth** will try to copy as many groups as can be held inside a *kauth_cred_t*.

void **kauth_cred_to_uucred**(*struct uucred *uucred*, *const kauth_cred_t cred*)
Convert *kauth_cred_t* to userland's view of credentials.

This includes effective user- and group-ids, a number of groups, and a group list.

Note that **kauth** will try to copy as many groups as can be held inside a *struct uucred*.

int **kauth_cred_uucmp**(*kauth_cred_t cred*, *struct uucred *uucred*)
Compares *cred* with the userland credentials in *uucred*.

Common values that will be compared are effective user- and group-ids, and the group list.

Miscellaneous Routines

Other routines provided by **kauth** are:

void **kauth_cred_clone**(*kauth_cred_t cred1*, *kauth_cred_t cred2*)
Clone credentials from *cred1* to *cred2*, except for the lock and reference count.

kauth_cred_t **kauth_cred_dup**(*kauth_cred_t cred*)
Duplicate *cred*.

What this routine does is call **kauth_cred_alloc**() followed by a call to **kauth_cred_clone**().

kauth_cred_t **kauth_cred_copy**(*kauth_cred_t cred*)
Works like **kauth_cred_dup**(), except for a few differences.

If *cred* already has a reference count of one, it will be returned. Otherwise, a new *kauth_cred_t* will be allocated and the credentials from *cred* will be cloned to it. Last, a call to **kauth_cred_free**() for *cred* will be done.

kauth_cred_t **kauth_cred_get**(*void*)
Return the credentials associated with the current LWP.

Scope Management

kauth provides routines to manage the creation and deletion of scopes on the system.

Note that the built-in scopes, the "generic" scope and the "process" scope, can't be deleted.

kauth_scope_t **kauth_register_scope**(*const char *id*, *kauth_scope_callback_t cb*, *void *cookie*)
Register a new scope on the system. *id* is the name of the scope, usually in reverse DNS-like

notation. For example, "org.netbsd.kauth.myscope". *cb* is the default listener, to which authorization requests for this scope will be dispatched to. *cookie* is optional user-data that will be passed to all listeners during authorization on the scope.

void **kauth_deregister_scope**(*kauth_scope_t scope*)
Deregister *scope* from the scopes available on the system, and free the *kauth_scope_t* object *scope*.

Listener Management

Listeners in **kauth** are authorization callbacks that are called during an authorization request in the scope which they belong to.

When an authorization request is made, all listeners associated with a scope are called to allow, deny, or defer the request.

It is enough for one listener to deny the request in order for the request to be denied; but all listeners are called during an authorization process none-the-less. All listeners are required to allow the request for it to be granted, and in a case where all listeners defer the request -- leaving the decision for other listeners -- the request is denied.

The following KPI is provided for the management of listeners:

kauth_listener_t **kauth_listen_scope**(*const char *id*, *kauth_scope_callback_t cb*, *void *cookie*)
Create a new listener on the scope with the id *id*, setting the default listener to *cb*. *cookie* is optional user-data that will be passed to the listener when called during an authorization request.

void **kauth_unlisten_scope**(*kauth_listener_t listener*)
Removes *listener* from the scope which it belongs to, ensuring it won't be called again, and frees the *kauth_listener_t* object *listener*.

kauth provides no means for synchronization within listeners. It is the the programmer's responsibility to make sure data used by the listener is properly locked during its use, as it can be accessed simultaneously from the same listener called multiple times. It is also the programmer's responsibility to do garbage collection after the listener, possibly freeing any allocated data it used.

The common method to do the above is by having a reference count to each listener. On entry to the listener, this reference count should be raised, and on exit -- lowered.

During the removal of a listener, first **kauth_scope_unlisten**() should be called to make sure the listener code will not be entered in the future. Then, the code should wait (possibly sleeping) until the reference count drops to zero. When that happens, it is safe to do the final cleanup.

Listeners might sleep, so no locks can be held when calling an authorization wrapper.

EXAMPLES

Older code had no abstraction of the security model, so most privilege checks looked like this:

```
if (suser(cred, &acflag) == 0)
        /* allow privileged operation */
```

Using the new interface, you must ask for a specific privilege explicitly. For example, checking whether it is possible to open a socket would look something like this:

```
if (kauth_authorize_network(cred, KAUTH_NETWORK_SOCKET,
    KAUTH_REQ_NETWORK_SOCKET_OPEN, PF_INET, SOCK_STREAM,
    IPPROTO_TCP) == 0)
        /* allow opening the socket */
```

Note that the *securelevel* implications were also integrated into the **kauth** framework so you don't have to note anything special in the call to the authorization wrapper, but rather just have to make sure the security model handles the request as you expect it to.

To do that you can just `grep`(1) in the relevant security model directory and have a look at the code.

EXTENDING KAUTH

Although **kauth** provides a large set of both detailed and more or less generic requests, it might be needed eventually to introduce more scopes, actions, or requests.

Adding a new scope should happen only when an entire subsystem is introduced and it is assumed other parts of the kernel may want to interfere with its inner-workings. When a subsystem that has the potential of impacting the security if the system is introduced, existing security modules must be updated to also handle actions on the newly added scope.

New actions should be added when sets of operations not covered at all belong in an already existing scope.

Requests (or sub-actions) can be added as subsets of existing actions when an operation that belongs in an already covered area is introduced.

Note that all additions should include updates to this manual, the security models shipped with NetBSD, and the example skeleton security model.

SEE ALSO

`secmodel`(9)

HISTORY

The kernel authorization framework first appeared in Mac OS X 10.4.

The kernel authorization framework in NetBSD first appeared in NetBSD 4.0, and is a clean-room implementation based on Apple TN2127, available at http://developer.apple.com/technotes/tn2005/tn2127.html

NOTES

As **kauth** in NetBSD is still under active development, it is likely that the ABI, and possibly the API, will differ between NetBSD versions. Developers are to take notice of this fact in order to avoid building code that expects one version of the ABI and running it in a system with a different one.

AUTHORS

Elad Efrat ⟨elad@NetBSD.org⟩ implemented the kernel authorization framework in NetBSD.

Jason R. Thorpe ⟨thorpej@NetBSD.org⟩ provided guidance and answered questions about the Darwin implementation.

ONE MORE THING

The **kauth** framework is dedicated to Brian Mitchell, one of the most talented people I know. Thanks for everything.

NAME

kcopy — copy data with abort on page fault

SYNOPSIS

```
#include <sys/systm.h>

int
kcopy(const void *src, void *dst, size_t len);
```

DESCRIPTION

kcopy() copies *len* bytes from *src* to *dst*, aborting if a fatal page fault is encountered.

kcopy() must save and restore the old fault handler since it is called by uiomove(9), which may be in the path of servicing a non-fatal page fault. **kcopy**() returns 0 on success and an error number on failure.

SEE ALSO

errno(2), memcpy(9), uiomove(9)

NAME

kfilter_register, **kfilter_unregister** — add or remove kernel event filters

SYNOPSIS

```
#include <sys/event.h>

int
kfilter_register(const char *name, struct filterops *filtops,
    int *retfilter);

int
kfilter_unregister(const char *name);
```

DESCRIPTION

The **kfilter_register**() function adds a new kernel event filter (kfilter) to the system, for use by callers of kqueue(2) and kevent(2). *name* is the name of the new filter (which must not already exist), and *filtops* is a pointer to a *filterops* structure which describes the filter operations. Both *name* and *filtops* will be copied to an internal data structure, and a new filter number will be allocated. If *retfilter* is not NULL, then the new filter number will be returned in the address pointed at by *retfilter*.

The **kfilter_unregister**() function removes a kfilter named *name* that was previously registered with **kfilter_register**(). If a filter with the same *name* is later reregistered with **kfilter_register**(), it will get a different filter number (i.e., filter numbers are not recycled). It is not possible to unregister the system filters (i.e., those that start with "EVFILT_" and are documented in kqueue(2)).

The *filterops* structure is defined as follows:

```
struct filterops {
        int     f_isfd;         /* true if ident == filedescriptor */
        int     (*f_attach)(struct knote *kn);
                                /* called when knote is ADDed */
        void    (*f_detach)(struct knote *kn);
                                /* called when knote is DELETEd */
        int     (*f_event)(struct knote *kn, long hint);
                                /* called when event is triggered */
};
```

If the filter operation is for a file descriptor, *f_isfd* should be non-zero, otherwise it should be zero. This controls where the kqueue(2) system stores the knotes for an object.

RETURN VALUES

kfilter_register() returns 0 on success, EINVAL if there's an invalid argument, or EEXIST if the filter already exists,

kfilter_unregister() returns 0 on success, EINVAL if there's an invalid argument, or ENOENT if the filter doesn't exist.

SEE ALSO

kqueue(2), free(9), knote(9), malloc(9)

HISTORY

The **kfilter_register**() and **kfilter_unregister**() functions first appeared in NetBSD 2.0.

AUTHORS

The **kfilter_register**() and **kfilter_unregister**() functions were implemented by Luke Mewburn ⟨lukem@NetBSD.org⟩.

NAME

kmem_alloc — allocate kernel wired memory

SYNOPSIS

```
#include <sys/kmem.h>

void *
kmem_alloc(size_t size, km_flag_t kmflags);
```

DESCRIPTION

kmem_alloc() allocates kernel wired memory. It takes the following arguments.

size Specify the size of allocation in bytes.

kmflags Either of the following:
KM_SLEEP Can sleep until enough memory is available.
KM_NOSLEEP
Don't sleep. Immediately return NULL if there is not enough memory available.

The contents of allocated memory are uninitialized.

Unlike Solaris, kmem_alloc(0, flags) is illegal.

RETURN VALUES

On success, **kmem_alloc**() returns a pointer to allocated memory. Otherwise, it returns NULL.

SEE ALSO

intro(9), kmem_free(9), kmem_zalloc(9), malloc(9), memoryallocators(9)

CAVEATS

kmem_alloc() can not be used from interrupt context.

SECURITY CONSIDERATION

As the allocated memory is uninitialized, it can contain security-sensitive data left by its previous user. It's the caller's responsibility not to expose it to the world.

NAME

kmem_free — free kernel wired memory

SYNOPSIS

```
#include <sys/kmem.h>

void
kmem_free(void *p, size_t size);
```

DESCRIPTION

kmem_free() frees kernel wired memory allocated by **kmem_alloc**() or **kmem_zalloc**() so that it can be used for other purposes. It takes the following arguments.

p The pointer to the memory being freed. It must be the one returned by **kmem_alloc**() or **kmem_zalloc**().

size The size of the memory being freed, in bytes. It must be the same as the *size* argument used for **kmem_alloc**() or **kmem_zalloc**() when the memory was allocated.

Freeing NULL is illegal.

SEE ALSO

intro(9), kmem_alloc(9), kmem_zalloc(9), memoryallocators(9)

CAVEATS

kmem_free() can not be used from interrupt context.

NAME

kmem_zalloc — allocate zero-initialized kernel wired memory

SYNOPSIS

```
#include <sys/kmem.h>

void *
kmem_zalloc(size_t size, km_flag_t kmflags);
```

DESCRIPTION

kmem_zalloc() is the equivalent of **kmem_alloc**(), except that it initializes the memory to zero.

SEE ALSO

intro(9), kmem_alloc(9), kmem_free(9), memoryallocators(9)

CAVEATS

kmem_zalloc() can not be used from interrupt context.

NAME

knote, **KNOTE** — raise kernel event

SYNOPSIS

```
#include <sys/event.h>

void
knote(struct klist *list, long hint);

KNOTE(struct klist *list, long hint);
```

DESCRIPTION

The **knote**() function provides a hook into the kqueue kernel event notification mechanism to allow sections of the kernel to raise a kernel event in the form of a 'knote', which is a *struct knote* as defined in ⟨sys/event.h⟩.

knote() takes a singly linked *list* of knotes, along with a *hint* (which is passed to the appropriate filter routine). **knote**() then walks the *list* making calls to the filter routine for each knote. As each knote contains a reference to the data structure that it is attached to, the filter may choose to examine the data structure in deciding whether an event should be reported. The *hint* is used to pass in additional information, which may not be present in the data structure that the filter examines.

If the filter decides that the event should be returned, it returns a non-zero value and **knote**() links the knote onto the tail end of the active list in the corresponding kqueue for the application to retrieve. If the knote is already on the active list, no action is taken, but the call to the filter occurs in order to provide an opportunity for the filter to record the activity.

knote() must not be called from interrupt contexts running at an interrupt priority level higher than **splsched**().

KNOTE() is a macro that calls **knote**(*list*, *hint*) if *list* is not empty.

SEE ALSO

kqueue(2), kfilter_register(9)

HISTORY

The **knote**() and **KNOTE**() functions first appeared in FreeBSD 4.1, and then in NetBSD 2.0.

AUTHORS

The **kqueue**() system was written by Jonathan Lemon ⟨jlemon@FreeBSD.org⟩.

NAME

kpause — make the calling LWP sleep

SYNOPSIS

```
#include <sys/proc.h>

int
kpause(const char *wmesg, bool intr, int timeo, kmutex_t *mtx);
```

DESCRIPTION

kpause() makes the calling LWP sleep. It's similar to cv_timedwait_sig(9) without the corresponding cv_signal(9).

kpause() can wake up spontaneously. Callers should prepare to handle it.

wmesg Specifies a string of no more than 8 characters that describes the resource or condition associated with the call of **kpause**(). The kernel does not use this argument directly but makes it available for utilities such as ps(1) to display.

intr If true, sleep interruptably. If the LWP recieves a signal, or is interrupted by another condition such as its containing process exiting, the wait is ended early and an error code returned.

timeo Specify a timeout. It is an architecture and system dependent value related to the number of clock interrupts per second. See hz(9) for details. The mstohz(9) macro can be used to convert a timeout expressed in milliseconds to one suitable for **kpause**().

Zero means no timeout.

mtx Convenience and symmetry with other synchronization operations. If not NULL, *mtx* will be released once the LWP has prepared to sleep, and will be reacquired before **kpause**() returns.

RETURN VALUES

kpause() returns 0 when waking up spontaneously. Otherwise, It returns an error number.

ERRORS

[EWOULDBLOCK] The timeout expired.

[ERESTART] **kpause**() returned as a result of a signal with SA_RESTART property.

[EINTR] **kpause**() returned due to other reasons. Typically as a result of a signal without SA_RESTART property.

SEE ALSO

sigaction(2), errno(9), condvar(9), mstohz(9), hz(9)

NAME

kpreempt — control kernel preemption

SYNOPSIS

```
#include <sys/systm.h>

void
kpreempt_disable(void);

void
kpreempt_enable(void);

bool
kpreempt_disabled(void);
```

DESCRIPTION

These functions are used to control kernel preemption of the calling LWP.

Kernel preemption is currently disabled by default. It can be enabled by tweaking kern.sched.kpreempt_pri sysctl.

kpreempt_disable() disables kernel preemption of the calling LWP. Note that disabling kernel preemption can prevent LWPs with higher priorities from running.

kpreempt_enable() enables kernel preemption of the calling LWP, which was previously disabled by **kpreempt_disable**().

kpreempt_disable() and **kpreempt_enable**() can be nested.

kpreempt_disabled() returns `true` if preemption of the calling LWP is disabled. It's for diagnostic purpose.

SEE ALSO

`intro`(9), `spl`(9)

NAME

printf, snprintf, vprintf, vsnprintf, uprintf, ttyprintf, tprintf, aprint — kernel formatted output conversion

SYNOPSIS

```
#include <sys/systm.h>

void
printf(const char *format, ...);

void
printf_nolog(const char *format, ...);

int
snprintf(char *buf, size_t size, const char *format, ...);

#include <machine/stdarg.h>

void
vprintf(const char *format, va_list ap);

int
vsnprintf(char *buf, size_t size, const char *format, va_list ap);

void
uprintf(const char *format, ...);

void
ttyprintf(struct tty *tty, const char *format, ...);

#include <sys/tprintf.h>

tpr_t
tprintf_open(struct proc *p);

void
tprintf(tpr_t tpr, const char *format, ...);

void
tprintf_close(tpr_t tpr);

void
aprint_normal(const char *format, ...);

void
aprint_naive(const char *format, ...);

void
aprint_verbose(const char *format, ...);

void
aprint_debug(const char *format, ...);

void
aprint_error(const char *format, ...);

void
aprint_normal_dev(device_t, const char *format, ...);

void
aprint_naive_dev(device_t, const char *format, ...);
```

void
aprint_verbose_dev(*device_t*, *const char *format*, *...*);

void
aprint_debug_dev(*device_t*, *const char *format*, *...*);

void
aprint_error_dev(*device_t*, *const char *format*, *...*);

void
aprint_normal_ifnet(*struct ifnet **, *const char *format*, *...*);

void
aprint_naive_ifnet(*struct ifnet **, *const char *format*, *...*);

void
aprint_verbose_ifnet(*struct ifnet **, *const char *format*, *...*);

void
aprint_debug_ifnet(*struct ifnet **, *const char *format*, *...*);

void
aprint_error_ifnet(*struct ifnet **, *const char *format*, *...*);

int
aprint_get_error_count(*void*);

DESCRIPTION

The **printf**() family of functions allows the kernel to send formatted messages to various output devices. The functions **printf**() and **vprintf**() send formatted strings to the system console. The **printf_nolog**() function is identical to **printf**(), except it does not send the data to the system log. The functions **snprintf**() and **vsnprintf**() write output to a string buffer. These four functions work similarly to their user space counterparts, and are not described in detail here.

The functions **uprintf**() and **ttyprintf**() send formatted strings to the current process's controlling tty and a specific tty, respectively.

The **tprintf**() function sends formatted strings to a process's controlling tty, via a handle of type tpr_t. This allows multiple write operations to the tty with a guarantee that the tty will be valid across calls. A handle is acquired by calling **tprintf_open**() with the target process as an argument. This handle must be closed with a matching call to **tprintf_close**().

The functions **aprint_normal**(), **aprint_naive**(), **aprint_verbose**(), **aprint_debug**(), and **aprint_error**() are intended to be used to print autoconfiguration messages, and change their behavior based on flags in the "boothowto" variable:

aprint_normal() Sends to the console unless AB_QUIET is set. Always sends to the log.

aprint_naive() Sends to the console only if AB_QUIET is set. Never sends to the log.

aprint_verbose() Sends to the console only if AB_VERBOSE is set. Always sends to the log.

aprint_debug() Sends to the console and the log only if AB_DEBUG is set.

aprint_error() Like **aprint_normal**(), but also keeps track of the number of times called. This allows a subsystem to report the number of errors that occurred during a quiet or silent initialization phase.

For the **aprint_***() functions there are two additional families of functions with the suffixes _dev and _ifnet which work like their counterparts without the suffixes, except that they take a *device_t* or

struct ifnet * respectively as first argument and prefix the log message with the corresponding device or interface name.

The **aprint_get_error_count**() function reports the number of errors and resets the counter to 0.

If AB_SILENT is set, none of the autoconfiguration message printing routines send output to the console. The AB_VERBOSE and AB_DEBUG flags override AB_SILENT.

RETURN VALUES

The **snprintf**() and **vsnprintf**() functions return the number of characters placed in the buffer *buf*. This is different to the user-space functions of the same name.

The **tprintf_open**() function returns NULL if no terminal handle could be acquired.

SEE ALSO

printf(1), printf(3), bitmask_snprintf(9)

CODE REFERENCES

sys/kern/subr_prf.c

HISTORY

The **sprintf**() and **vsprintf**() unsized string formatting functions are supported for compatibility only, and are not documented here. New code should use the size-limited **snprintf**() and **vsnprintf**() functions instead.

In NetBSD 1.5 and earlier, **printf**() supported more format strings than the user space **printf**(). These nonstandard format strings are no longer supported. For the functionality provided by the former %b format string, see bitmask_snprintf(9).

The **aprint_normal**(), **aprint_naive**(), **aprint_verbose**(), and **aprint_debug**() functions first appeared in BSD/OS.

BUGS

The **uprintf**() and **ttyprintf**() functions should be used sparingly, if at all. Where multiple lines of output are required to reach a process's controlling terminal, **tprintf**() is preferred.

NAME

kthread_create, **kthread_destroy**, **kthread_exit** — kernel threads

SYNOPSIS

```
#include <sys/kthread.h>

int
kthread_create(pri_t pri, int flags, struct cpu_info *ci,
    void (*func)(void *), void *arg, lwp_t **newlp, const char *fmt, ...);

void
kthread_destroy(lwp_t *l);

void
kthread_exit(int ecode);
```

DESCRIPTION

Kernel threads are light-weight processes which execute entirely within the kernel.

Any process can request the creation of a new kernel thread. Kernel threads are not swapped out during memory congestion. The VM space and limits are shared with proc0 (usually swapper).

FUNCTIONS

kthread_create(*pri*, *flags*, *ci*, *func*, *arg*, *newlp*, *fmt*, ...)

Create a kernel thread. The arguments are as follows.

pri Priority level for the thread. If no priority level is desired specify PRI_NONE, causing **kthread_create**() to select the default priority level.

flags Flags that can be logically ORed together to alter the thread's behaviour.

KTHREAD_IDLE: causes the thread to be created in the LSIDL (idle) state. By default, the threads are created in the LSRUN (runnable) state, meaning they will begin execution shortly after creation.

KTHREAD_MPSAFE: Specifies that the thread does its own locking and so is multiprocessor safe. If not specified, the global kernel lock will be held whenever the thread is running (unless explicitly dropped by the thread).

KTHREAD_INTR: Specifies that the thread services device interrupts. This flag is intended for kernel internal use and should not normally be specified.

ci If non-NULL, the thread will be created bound to the CPU specified by *ci*, meaning that it will only ever execute on that CPU. By default, the threads are free to execute on any CPU in the system.

func A function to be called when the thread begins executing. This function must not return. If the thread runs to completion, it must call **kthread_exit**() to properly terminate itself.

arg An argument to be passed to **func**(). May be NULL if not required.

newpl A pointer to receive the new lwp structure for the kernel thread. May be NULL if not required.

fmt A string containing format information used to display the kernel thread name. Must not be NULL.

kthread_destroy(*l*)
From another thread executing in the kernel, cause a kthread to exit. The kthread must be in the LSIDL (idle) state.

kthread_exit(*ecode*)
Exit from a kernel thread. Must only be called by a kernel thread.

RETURN VALUES

Upon successful completion, **kthread_create**() returns 0. Otherwise, the following error values are returned:

[EAGAIN] The limit on the total number of system processes would be exceeded.

[EAGAIN] The limit RLIMIT_NPROC on the total number of processes under execution by this user id would be exceeded.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the kthread framework can be found. All pathnames are relative to /usr/src.

The kthread framework itself is implemented within the file sys/kern/kern_kthread.c. Data structures and function prototypes for the framework are located in sys/sys/kthread.h.

SEE ALSO

driver(9)

HISTORY

The kthread framework appeared in NetBSD 1.4.

NAME

ttyldisc_add, **ttyldisc_lookup**, **ttyldisc_remove** — extensible line discipline framework

SYNOPSIS

```
#include <sys/conf.h>

int
ttyldisc_add(struct linesw *disc, int no);

struct linesw *
ttyldisc_remove(const char *name);

struct linesw *
ttyldisc_lookup(const char *name);
```

DESCRIPTION

The NetBSD TTY line discipline framework allows extensibility. Modules that need special line disciplines can add them as convenient and do not need to modify tty_conf.c. Line disciplines are now managed by a string, rather than number.

Once the framework has been initialized, a new line discipline can be added by creating and initializing a *struct linesw* and calling **ttyldisc_add**().

The following is a brief description of each function in the framework:

ttyldisc_add() Register a line discipline. The *l_name* field of the *struct linesw* should point to a string which is to be the symbolic name of that line discipline. For compatibility purposes, a line discipline number can be passed in *no*, but for new disciplines this should be set to `-1`.

ttyldisc_lookup() Look up a line discipline by *name*. `NULL` is returned if it can not be found.

ttyldisc_remove() Remove a line discipline called *name* and return a pointer to it. If the discipline cannot be found or removed **ttyldisc_remove**() will return `NULL`.

SEE ALSO

`tty`(4)

HISTORY

The **ttyldisc_add** functions were added in NetBSD 1.6.

AUTHORS

The NetBSD extensible line discipline framework was created by Eduardo Horvath ⟨eeh@NetBSD.org⟩.

NAME

lock, **simple_lock_init**, **simple_lock**, **simple_lock_try**, **simple_unlock**, **simple_lock_freecheck**, **simple_lock_dump**, **lockinit**, **lockmgr**, **lockstatus**, **lockmgr_printinfo**, **spinlockinit**, **spinlockmgr** — kernel lock functions

DESCRIPTION

These interfaces have been obsoleted and removed from the system.

Please see the condvar(9), mutex(9), and rwlock(9) manual pages for information on kernel synchronisation primitives.

SEE ALSO

condvar(9), mutex(9), rwlock(9)

HISTORY

The kernel locking API first appeared in 4.4BSD-lite2, and was replaced in NetBSD 5.0.

NAME

log — log a message from the kernel through the /dev/klog device

SYNOPSIS

```
#include <sys/syslog.h>

void
log(int level, const char *format, ...);
```

DESCRIPTION

The **log**() function allows the kernel to send messages to user processes listening on /dev/klog. Usually syslogd(8) monitors /dev/klog for these messages and writes them to a log file.

All messages are logged using facility LOG_KERN. See syslog(3) for a listing of log levels.

SEE ALSO

syslog(3), syslogd(8)

NAME

ltsleep, **tsleep**, **wakeup** — process context sleep and wakeup

SYNOPSIS

```
#include <sys/proc.h>

int
ltsleep(wchan_t ident, pri_t priority, const char *wmesg, int timo,
    volatile struct simplelock *slock);

int
tsleep(wchan_t ident, pri_t priority, const char *wmesg, int timo);

void
wakeup(wchan_t ident);
```

DESCRIPTION

The interfaces described in this manual page are obsolete and will be removed from a future version of the system.

Please see the condvar(9), mutex(9), *and* rwlock(9) *manual pages for information on kernel synchronisation primitives.*

These functions implement voluntary context switching. **ltsleep**() and **tsleep**() are used throughout the kernel whenever processing in the current context can not continue for any of the following reasons:

- The current process needs to await the results of a pending I/O operation.
- The current process needs resources (e.g., memory) which are temporarily unavailable.
- The current process wants access to data-structures which are locked by other processes.

The function **wakeup**() is used to notify sleeping processes of possible changes to the condition that caused them to go to sleep. Typically, an awakened process will -- after it has acquired a context again -- retry the action that blocked its operation to see if the "blocking" condition has cleared.

The **ltsleep**() function takes the following arguments:

ident An identifier of the "wait channel" representing the resource for which the current process needs to wait. This typically is the virtual address of some kernel data-structure related to the resource for which the process is contending. The same identifier must be used in a call to **wakeup**() to get the process going again. *ident* should not be NULL.

priority The process priority to be used when the process is awakened and put on the queue of runnable processes. This mechanism is used to optimize "throughput" of processes executing in kernel mode. If the flag PCATCH is OR'ed into *priority* the process checks for posted signals before and after sleeping. If the flag PNORELOCK is OR'ed into *priority*, *slock* is NOT re-locked after process resume.

wmesg A pointer to a character string indicating the reason a process is sleeping. The kernel does not use the string, but makes it available (through the process structure field p_wmesg) for user level utilities such as ps(1).

timo If non-zero, the process will sleep for at most timo/hz seconds. If this amount of time elapses and no **wakeup**(*ident*) has occurred, and no signal (if PCATCH was set) was posted, **tsleep**() will return EWOULDBLOCK.

slock If not NULL, the *slock* interlock is unlocked once the scheduler lock is acquired. Unless PNORELOCK was set, *slock* is locked again once the process is resumed from sleep. This provides wakeup-before-sleep condition protection facility.

The **tsleep**() macro is functionally equivalent to:

```
ltsleep(ident, priority, wmesg, timo, NULL)
```

The **wakeup**() function will mark all processes which are currently sleeping on the identifier *ident* as runnable. Eventually, each of the processes will resume execution in the kernel context, causing a return from **tsleep**(). Note that processes returning from sleep should always re-evaluate the conditions that blocked them, since a call to **wakeup**() merely signals a *possible* change to the blocking conditions. For example, when two or more processes are waiting for an exclusive-access lock (see lock(9)), only one of them will succeed in acquiring the lock when it is released. All others will have to go back to sleep and wait for the next opportunity.

RETURN VALUES

ltsleep() returns 0 if it returns as a result of a **wakeup**(). If a **ltsleep**() returns as a result of a signal, the return value is ERESTART if the signal has the SA_RESTART property (see sigaction(2)), and EINTR otherwise. If **ltsleep**() returns because of a timeout it returns EWOULDBLOCK.

SEE ALSO

sigaction(2), condvar(9), hz(9), lock(9), mutex(9), rwlock(9)

HISTORY

The sleep/wakeup process synchronization mechanism is very old. It appeared in a very early version of Unix. **tsleep**() appeared in 4.4BSD. **ltsleep**() appeared in NetBSD 1.5.

NAME

m_tag, **m_tag_get**, **m_tag_free**, **m_tag_prepend**, **m_tag_unlink**, **m_tag_delete**, **m_tag_delete_chain**, **m_tag_delete_nonpersistent**, **m_tag_find**, **m_tag_copy**, **m_tag_copy_chain**, **m_tag_init**, **m_tag_first**, **m_tag_next** — mbuf tagging interfaces

SYNOPSIS

```
#include <sys/mbuf.h>

struct m_tag *
m_tag_get(int type, int len, int wait);

void
m_tag_free(struct m_tag *t);

void
m_tag_prepend(struct mbuf *m, struct m_tag *t);

void
m_tag_unlink(struct mbuf *m, struct m_tag *t);

void
m_tag_delete(struct mbuf *m, struct m_tag *t);

void
m_tag_delete_chain(struct mbuf *m, struct m_tag *t);

void
m_tag_delete_nonpersistent(struct mbuf *);

struct m_tag *
m_tag_find(struct mbuf *m, int type, struct m_tag *t);

struct m_tag *
m_tag_copy(struct m_tag *m);

int
m_tag_copy_chain(struct mbuf *to, struct mbuf *from);

void
m_tag_init(struct mbuf *m);

struct m_tag *
m_tag_first(struct mbuf *m);

struct m_tag *
m_tag_next(struct mbuf *m, struct m_tag *t);
```

DESCRIPTION

The **m_tag** interface is used to "tag" mbufs.

FUNCTIONS

m_tag_get(*type*, *len*, *wait*)
: Allocate an mbuf tag. *type* is one of the PACKET_TAG_ macros. *len* is the size of the data associated with the tag, in bytes. *wait* is either M_WAITOK or M_NOWAIT.

m_tag_free(*t*)
: Free the mbuf tag *t*.

m_tag_prepend(*m*, *t*)
Prepend the mbuf tag *t* to the mbuf *m*. *t* will become the first tag of the mbuf *m*. When *m* is freed, *t* will also be freed.

m_tag_unlink(*m*, *t*)
Unlink the mbuf tag *t* from the mbuf *m*.

m_tag_delete(*m*, *t*)
The same as **m_tag_unlink**() followed by **m_tag_free**().

m_tag_delete_chain(*m*, *t*)
Unlink and free mbuf tags beginning with the mbuf tag *t* from the mbuf *m*. If *t* is NULL, **m_tag_delete_chain**() unlinks and frees all mbuf tags associated with the mbuf *m*.

m_tag_delete_nonpersistent(*m*)
Unlink and free all non persistent tags associated with the mbuf *m*.

m_tag_find(*m*, *type*, *t*)
Find an mbuf tag with type *type* after the mbuf tag *t* in the tag chain associated with the mbuf *m*. If *t* is NULL, search from the first mbuf tag. If an mbuf tag is found, return a pointer to it. Otherwise return NULL.

m_tag_copy(*t*)
Copy an mbuf tag *t*. Return a new mbuf tag on success. Otherwise return NULL.

m_tag_copy_chain(*to*, *from*)
Copy all mbuf tags associated with the mbuf *from* to the mbuf *to*. If *to* already has any mbuf tags, they will be unlinked and freed beforehand. Return 1 on success. Otherwise return 0.

m_tag_init(*m*)
Initialize mbuf tag chain of the mbuf *m*.

m_tag_first(*m*)
Return the first mbuf tag associated with the mbuf *m*. Return NULL if no mbuf tags are found.

m_tag_next(*m*, *t*)
Return the next mbuf tag after *t* associated with the mbuf *m*. Return NULL if *t* is the last tag in the chain.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing the mbuf tagging interfaces can be found. All pathnames are relative to /usr/src.

The mbuf tagging interfaces are implemented within the file sys/kern/uipc_mbuf2.c.

The PACKET_TAG_ macros are defined in the file sys/sys/mbuf.h.

SEE ALSO

intro(9), malloc(9), mbuf(9)

BUGS

The semantics of the term "persistent tag" are vague.

NAME

makeiplcookie — compose an ipl cookie

SYNOPSIS

```
#include <sys/param.h>

ipl_cookie_t
makeiplcookie(ipl_t ipl);
```

DESCRIPTION

makeiplcookie() composes a cookie which can be fed into *splraiseipl*. *ipl* should be one of IPL_ constants.

RETURN VALUES

makeiplcookie() returns a composed cookie.

SEE ALSO

spl(9), splraiseipl(9)

NAME

malloc, **MALLOC**, **realloc**, **free**, **FREE**, **malloc_roundup**, **malloc_type_attach**, **malloc_type_detach**, **malloc_type_setlimit**, **MALLOC_DEFINE_LIMIT**, **MALLOC_DEFINE**, **MALLOC_DECLARE** — general-purpose kernel memory allocator

SYNOPSIS

```
#include <sys/malloc.h>

void *
malloc(unsigned long size, struct malloc_type *type, int flags);

MALLOC(space, cast, unsigned long size, struct malloc_type *type, int flags);

void *
realloc(void *addr, unsigned long newsize, struct malloc_type *type,
    int flags);

void
free(void *addr, struct malloc_type *type);

FREE(void *addr, struct malloc_type *type);

unsigned long
malloc_roundup(unsigned long size);

void
malloc_type_attach(struct malloc_type *type);

void
malloc_type_detach(struct malloc_type *type);

void
malloc_type_setlimit(struct malloc_type *type, unsigned long limit);

#include <sys/mallocvar.h>

MALLOC_DEFINE_LIMIT(type, shortdesc, longdesc, limit);

MALLOC_JUSTDEFINE_LIMIT(type, shortdesc, longdesc, limit);

MALLOC_DEFINE(type, shortdesc, longdesc);

MALLOC_JUSTDEFINE(type, shortdesc, longdesc);

MALLOC_DECLARE(type);
```

DESCRIPTION

These interfaces are being obsoleted and their new use is discouraged. For new code, please consider to use kmem_alloc(**9**) **or** pool_cache(**9**) **instead.**

The **malloc**() function allocates uninitialized memory in kernel address space for an object whose size is specified by *size*. **malloc_roundup**() returns the actual size of the allocation unit for the given value. **free**() releases memory at address *addr* that was previously allocated by **malloc**() for re-use. Unlike free(3), **free**() does not accept an *addr* argument that is NULL.

The **realloc**() function changes the size of the previously allocated memory referenced by *addr* to *size* and returns a pointer to the (possibly moved) object. The memory contents are unchanged up to the lesser of the new and old sizes. If the new size is larger, the newly allocated memory is uninitialized. If the requested memory cannot be allocated, NULL is returned and the memory referenced by *addr* is unchanged. If *addr* is NULL, then **realloc**() behaves exactly as **malloc**(). If the new size is 0, then **realloc**()

behaves exactly as **free**().

The **MALLOC**() macro variant is functionally equivalent to

```
(space) = (cast)malloc((u_long)(size), type, flags)
```

and the **FREE**() macro variant is equivalent to

```
free((void *)(addr), type)
```

The **MALLOC**() macro is intended to be used with a compile-time constant *size* so that the compiler can do constant folding. In the comparison to **malloc**() and **free**() functions, the **MALLOC**() and **FREE**() macros may be faster, at the cost of increased code size. There is no difference between the memory allocated with MALLOC and malloc. i.e., no matter which MALLOC or malloc is used to allocate the memory, either FREE or free can be used to free it.

Unlike its standard C library counterpart (malloc(3)), the kernel version takes two more arguments.

The *flags* argument further qualifies **malloc**() operational characteristics as follows:

M_NOWAIT Causes **malloc**() to return NULL if the request cannot be immediately fulfilled due to resource shortage. If this flag is not set (see M_WAITOK), **malloc**() will never return NULL.

M_WAITOK By default, **malloc**() may call cv_wait(9) to wait for resources to be released by other processes, and this flag represents this behaviour. Note that M_WAITOK is conveniently defined to be 0, and hence may be or'ed into the *flags* argument to indicate that it's ok to wait for resources.

M_ZERO Causes the allocated memory to be set to all zeros.

M_CANFAIL Changes behaviour for M_WAITOK case - if the requested memory size is bigger than **malloc**() can ever allocate, return failure, rather than calling panic(9). This is different to M_NOWAIT, since the call can still wait for resources.

Rather than depending on M_CANFAIL, kernel code should do proper bound checking itself. This flag should only be used in cases where this is not feasible. Since it can hide real kernel bugs, its usage is *strongly discouraged*.

The *type* argument describes the subsystem and/or use within a subsystem for which the allocated memory was needed, and is commonly used to maintain statistics about kernel memory usage and, optionally, enforce limits on this usage for certain memory types.

In addition to some built-in generic types defined by the kernel memory allocator, subsystems may define their own types.

The **MALLOC_DEFINE_LIMIT**() macro defines a malloc type named *type* with the short description *shortdesc*, which must be a constant string; this description will be used for kernel memory statistics reporting. The *longdesc* argument, also a constant string, is intended as way to place a comment in the actual type definition, and is not currently stored in the type structure. The *limit* argument specifies the maximum amount of memory, in bytes, that this malloc type can consume.

The **MALLOC_DEFINE**() macro is equivalent to the **MALLOC_DEFINE_LIMIT**() macro with a *limit* argument of 0. If kernel memory statistics are being gathered, the system will choose a reasonable default limit for the malloc type.

The **MALLOC_DECLARE**() macro is intended for use in header files which are included by code which needs to use the malloc type, providing the necessary extern declaration.

Code which includes <sys/malloc.h> does not need to include <sys/mallocvar.h> to get these macro definitions. The <sys/mallocvar.h> header file is intended for other header files which need to use the **MALLOC_DECLARE**() macro.

The **malloc_type_attach**() function attaches the malloc type *type* to the kernel memory allocator. This is intended for use by LKMs; malloc types included in modules statically-linked into the kernel are automatically registered with the kernel memory allocator. However, it is possible to define malloc types without automatically registering them using **MALLOC_JUSTDEFINE**() or **MALLOC_JUSTDEFINE_LIMIT**(). Apart from not automatically registering to the kernel a boot time, these functions are equivalent to their counterparts. They can be used when a separate LKM codepath for initialization is not desired.

The **malloc_type_detach**() function detaches the malloc type *type* previously attached with **malloc_type_attach**().

The **malloc_type_setlimit**() function sets the memory limit of the malloc type *type* to *limit* bytes. The type must already be registered with the kernel memory allocator.

The following generic malloc types are currently defined:

M_DEVBUF	Device driver memory.
M_DMAMAP	bus_dma(9) structures.
M_FREE	Should be on free list.
M_PCB	Protocol control block.
M_SOFTINTR	Softinterrupt structures.
M_TEMP	Misc temporary data buffers.

Other malloc types are defined by the corresponding subsystem; see the documentation for that subsystem for information its available malloc types.

Statistics based on the *type* argument are maintained only if the kernel option KMEMSTATS is used when compiling the kernel (the default in current NetBSD kernels) and can be examined by using 'vmstat -m'.

RETURN VALUES

malloc() returns a kernel virtual address that is suitably aligned for storage of any type of object.

DIAGNOSTICS

A kernel compiled with the DIAGNOSTIC configuration option attempts to detect memory corruption caused by such things as writing outside the allocated area and imbalanced calls to the **malloc**() and **free**() functions. Failing consistency checks will cause a panic or a system console message:

- panic: "malloc - bogus type"
- panic: "malloc: out of space in kmem_map"
- panic: "malloc: allocation too large"
- panic: "malloc: wrong bucket"
- panic: "malloc: lost data"
- panic: "free: unaligned addr"
- panic: "free: duplicated free"
- panic: "free: multiple frees"
- panic: "init: minbucket too small/struct freelist too big"
- "multiply freed item ⟨addr⟩"
- "Data modified on freelist: ⟨data object description⟩"

SEE ALSO

`vmstat`(1), `memoryallocators`(9)

NAME

mb, **mb_memory**, **mb_read**, **mb_write** — memory barriers

SYNOPSIS

```
#include <sys/lock.h>

void
mb_memory(void);

void
mb_read(void);

void
mb_write(void);
```

DESCRIPTION

Many types of processor can execute instructions in a different order than issued by the compiler or assembler. On a uniprocessor system, out of order execution is transparent to the programmer, operating system and applications, as the processor must ensure that it is self consistent.

On multiprocessor systems, out of order execution can present a problem where locks are not used to guarantee atomicity of access, because loads and stores issued by any given processor can appear on the system bus (and thus appear to other processors) in an unpredictable order.

mb_memory(), **mb_read**(), and **mb_write**() can be used to control the order in which memory accesses occur, and thus the order in which those accesses become visible to other processors. They can be used to implement "lockless" access to data structures where the necessary barrier conditions are well understood.

Memory barriers can be computationally expensive, as they are considered "serializing" operations and may stall further execution until the processor has drained internal buffers and re-synchronized.

The memory barrier primitives control only the order of memory access. They provide no guarantee that stores have been flushed to the bus, or that loads have been made from the bus.

The memory barrier primitives are guaranteed only to prevent reordering of accesses to main memory. They do not provide any guarantee of ordering when used with device memory (for example, loads or stores to or from a PCI device). To guarantee ordering of access to device memory, the bus_dma(9) and bus_space(9) interfaces should be used.

FUNCTIONS

mb_memory()

Issue a full memory barrier, ordering all memory accesses. Causes all loads and stores preceding the call to **mb_memory**() to complete before further memory accesses can be made.

mb_read()

Issue a read memory barrier, ordering all loads from memory. Causes all loads preceding the call to **mb_read**() to complete before further loads can be made. Stores may be reordered ahead of or behind a call to **mb_read**().

mb_write()

Issue a write memory barrier, ordering all stores to memory. Causes all stores preceding the call to **mb_write**() to complete before further stores can be made. Loads may be reordered ahead of or behind a call to **mb_write**().

SEE ALSO

`bus_dma`(9), `bus_space`(9), `mutex`(9), `rwlock`(9)

HISTORY

The memory barrier primitives first appeared in NetBSD 5.0.

NAME

mbuf, **m_get**, **m_getclr**, **m_gethdr**, **m_devget**, **m_copym**, **m_copypacket**, **m_copydata**, **m_copyback**, **m_copyback_cow**, **m_cat**, **m_dup**, **m_makewritable**, **m_prepend**, **m_pulldown**, **m_pullup**, **m_split**, **m_adj**, **m_apply**, **m_free**, **m_freem**, **mtod**, **MGET**, **MGETHDR**, **MEXTMALLOC**, **MEXTADD**, **MCLGET**, **M_COPY_PKTHDR**, **M_MOVE_PKTHDR**, **M_ALIGN**, **MH_ALIGN**, **M_LEADINGSPACE**, **M_TRAILINGSPACE**, **M_PREPEND**, **MCHTYPE**, **MFREE** — functions and macros for managing memory used by networking code

SYNOPSIS

```
#include <sys/mbuf.h>

struct mbuf *
m_get(int nowait, int type);

struct mbuf *
m_getclr(int nowait, int type);

struct mbuf *
m_gethdr(int nowait, int type);

struct mbuf *
m_devget(char *buf, int totlen, int off0, struct ifnet *ifp,
    void (*copy)(const void *, void *, size_t));

struct mbuf *
m_copym(struct mbuf *m, int off0, int len, int wait);

struct mbuf *
m_copypacket(struct mbuf *m, int how);

void
m_copydata(struct mbuf *m, int off, int len, void *cp);

void
m_copyback(struct mbuf *m0, int off, int len, void *cp);

struct mbuf *
m_copyback_cow(struct mbuf *m0, int off, int len, void *cp, int how);

int
m_makewritable(struct mbuf **mp, int off, int len, int how);

void
m_cat(struct mbuf *m, struct mbuf *n);

struct mbuf *
m_dup(struct mbuf *m, int off0, int len, int wait);

struct mbuf *
m_prepend(struct mbuf *m, int len, int how);

struct mbuf *
m_pulldown(struct mbuf *m, int off, int len, int *offp);

struct mbuf *
m_pullup(struct mbuf *n, int len);

struct mbuf *
m_split(struct mbuf *m0, int len0, int wait);
```

void
m_adj(*struct mbuf *mp*, *int req_len*);

int
m_apply(*struct mbuf *m*, *int off*, *int len*,
*int *f(void *, void *, unsigned int)*, *void *arg*);

*struct mbuf **
m_free(*struct mbuf *m*);

void
m_freem(*struct mbuf *m*);

datatype
mtod(*struct mbuf *m*, *datatype*);

void
MGET(*struct mbuf *m*, *int how*, *int type*);

void
MGETHDR(*struct mbuf *m*, *int how*, *int type*);

void
MEXTMALLOC(*struct mbuf *m*, *int len*, *int how*);

void
MEXTADD(*struct mbuf *m*, *void *buf*, *int size*, *int type*,
*void (*free)(struct mbuf *, void *, size_t, void *)*, *void *arg*);

void
MCLGET(*struct mbuf *m*, *int how*);

void
M_COPY_PKTHDR(*struct mbuf *to*, *struct mbuf *from*);

void
M_MOVE_PKTHDR(*struct mbuf *to*, *struct mbuf *from*);

void
M_ALIGN(*struct mbuf *m*, *int len*);

void
MH_ALIGN(*struct mbuf *m*, *int len*);

int
M_LEADINGSPACE(*struct mbuf *m*);

int
M_TRAILINGSPACE(*struct mbuf *m*);

void
M_PREPEND(*struct mbuf *m*, *int plen*, *int how*);

void
MCHTYPE(*struct mbuf *m*, *int type*);

void
MFREE(*struct mbuf *m*, *struct mbuf *n*);

DESCRIPTION

The **mbuf** functions and macros provide an easy and consistent way to handle a networking stack's memory management needs. An **mbuf** consists of a header and a data area. It is of a fixed size, `MSIZE` (defined in ⟨`machine/param.h`⟩), which includes overhead. The header contains a pointer to the next **mbuf** in the 'mbuf chain', a pointer to the next 'mbuf chain', a pointer to the data area, the amount of data in this mbuf, its type and a `flags` field.

The `type` variable can signify:

`MT_FREE`	the mbuf should be on the ''free'' list
`MT_DATA`	data was dynamically allocated
`MT_HEADER`	data is a packet header
`MT_SONAME`	data is a socket name
`MT_SOOPTS`	data is socket options
`MT_FTABLE`	data is the fragment reassembly header
`MT_CONTROL`	mbuf contains ancillary (protocol control) data
`MT_OOBDATA`	mbuf contains out-of-band data.

The `flags` variable contains information describing the **mbuf**, notably:

`M_EXT`	has external storage
`M_PKTHDR`	is start of record
`M_EOR`	is end of record
`M_CLUSTER`	external storage is a cluster.

If an **mbuf** designates the start of a record (`M_PKTHDR`), its `flags` field may contain additional information describing the content of the record:

`M_BCAST`	sent/received as link-level broadcast
`M_MCAST`	sent/received as link-level multicast
`M_LINK0`, `M_LINK1`, `M_LINK2`	three link-level specific flags.

An **mbuf** may add a single 'mbuf cluster' of `MCLBYTES` bytes (also defined in ⟨`machine/param.h`⟩), which has no additional overhead and is used instead of the internal data area; this is done when at least `MINCLSIZE` bytes of data must be stored.

When the `M_EXT` flag is raised for an mbuf, the external storage area could be shared among multiple mbufs. Be careful when you attempt to overwrite the data content of the mbuf.

m_get(*int nowait*, *int type*)
: Allocates an mbuf and initializes it to contain internal data. The *nowait* parameter is a choice of `M_WAIT / M_DONTWAIT` from caller. `M_WAIT` means the call cannot fail, but may take forever. The *type* parameter is an mbuf type.

m_getclr(*int nowait*, *int type*)
: Allocates an mbuf and initializes it to contain internal data, then zeros the data area. The *nowait* parameter is a choice of `M_WAIT / M_DONTWAIT` from caller. The *type* parameter is an mbuf type.

m_gethdr(*int nowait*, *int type*)
: Allocates an mbuf and initializes it to contain a packet header and internal data. The *nowait* parameter is a choice of `M_WAIT / M_DONTWAIT` from caller. The *type* parameter is an mbuf type.

m_devget(*char *buf*, *int totlen*, *int off0*, *struct ifnet *ifp*, *void (*copy)(const void *, void *, size_t)*)
: Copies *len* bytes from device local memory into mbufs using copy routine *copy*. If parameter

off is non-zero, the packet is supposed to be trailer-encapsulated and *off* bytes plus the type and length fields will be skipped before copying. Returns the top of the mbuf chain it created.

m_copym(*struct mbuf *m*, *int off0*, *int len*, *int wait*)
Creates a copy of an mbuf chain starting *off0* bytes from the beginning, continuing for *len* bytes. If the *len* requested is M_COPYALL, the complete mbuf chain will be copied. The *wait* parameter is a choice of M_WAIT / M_DONTWAIT from caller.

m_copypacket(*struct mbuf *m*, *int how*)
Copies an entire packet, including header (which must be present). This function is an optimization of the common case m_copym(m, 0, M_COPYALL, *how*).

m_copydata(*struct mbuf *m*, *int off*, *int len*, *void *cp*)
Copies *len* bytes data from mbuf chain *m* into the buffer *cp*, starting *off* bytes from the beginning.

m_copyback(*struct mbuf *m0*, *int off*, *int len*, *void *cp*)
Copies *len* bytes data from buffer *cp* back into the mbuf chain *m0*, starting *off* bytes from the beginning of the chain, extending the mbuf chain if necessary. **m_copyback**() can only fail when extending the chain. The caller should check for this kind of failure by checking the resulting length of the chain in that case. It is an error to use **m_copyback**() on read-only mbufs.

m_copyback_cow(*struct mbuf *m0*, *int off*, *int len*, *void *cp*, *int how*)
Copies *len* bytes data from buffer *cp* back into the mbuf chain *m0* as **m_copyback**() does. Unlike **m_copyback**(), it is safe to use **m_copyback_cow**() on read-only mbufs. If needed, **m_copyback_cow**() automatically allocates new mbufs and adjusts the chain. On success, it returns a pointer to the resulting mbuf chain, and frees the original mbuf *m0*. Otherwise, it returns NULL. The *how* parameter is a choice of M_WAIT / M_DONTWAIT from the caller. Unlike **m_copyback**(), extending the mbuf chain isn't supported. It is an error to attempt to extend the mbuf chain using **m_copyback_cow**().

m_makewritable(*struct mbuf **mp*, *int off*, *int len*, *int how*)
Rearranges an mbuf chain so that *len* bytes from offset *off* are writable. When it meets read-only mbufs, it allocates new mbufs, adjusts the chain as **m_copyback_cow**() does, and copies the original content into them. **m_makewritable**() does *not* guarantee that all *len* bytes at *off* are consecutive. The *how* parameter is a choice of M_WAIT / M_DONTWAIT from the caller. **m_makewritable**() preserves the contents of the mbuf chain even in the case of failure. It updates a pointer to the mbuf chain pointed to by *mp*. It returns 0 on success. Otherwise, it returns an error code, typically ENOBUFS.

m_cat(*struct mbuf *m*, *struct mbuf *n*)
Concatenates mbuf chain *n* to *m*. Both chains must be of the same type; packet headers will *not* be updated if present.

m_dup(*struct mbuf *m*, *int off0*, *int len*, *int wait*)
Similarly to **m_copym**(), the function creates a copy of an mbuf chain starting *off0* bytes from the beginning, continuing for *len* bytes. While **m_copym**() tries to share external storage for mbufs with M_EXT flag, **m_dup**() will deep-copy the whole data content into new mbuf chain and avoids shared external storage.

m_prepend(*struct mbuf *m*, *int len*, *int how*)
Lesser-used path for **M_PREPEND**(): allocates new mbuf *m* of size *len* to prepend to the chain, copying junk along. The *how* parameter is a choice of M_WAIT / M_DONTWAIT from caller.

m_pulldown(*struct mbuf *m*, *int off*, *int len*, *int *offp*)
Rearranges an mbuf chain so that *len* bytes from offset *off* are contiguous and in the data area of an mbuf. The return value points to an mbuf in the middle of the mbuf chain *m*. If we call the

return value *n*, the contiguous data region is available at `mtod(n, void *) + *offp`, or `mtod(n, void *)` if *offp* is `NULL`. The top of the mbuf chain *m*, and mbufs up to *off*, will not be modified. On successful return, it is guaranteed that the mbuf pointed to by *n* does not have a shared external storage, therefore it is safe to update the contiguous region. Returns `NULL` and frees the mbuf chain on failure. *len* must be smaller or equal than `MCLBYTES`.

m_pullup(*struct mbuf *m*, *int len*)
Rearranges an mbuf chain so that *len* bytes are contiguous and in the data area of an mbuf (so that **mtod**() will work for a structure of size *len*). Returns the resulting mbuf chain on success, frees it and returns `NULL` on failure. If there is room, it will add up to `max_protohdr - len` extra bytes to the contiguous region to possibly avoid being called again. *len* must be smaller or equal than `MHLEN`.

m_split(*struct mbuf *m0*, *int len0*, *int wait*)
Partitions an mbuf chain in two pieces, returning the tail, which is all but the first *len0* bytes. In case of failure, it returns `NULL` and attempts to restore the chain to its original state.

m_adj(*struct mbuf *mp*, *int req_len*)
Shaves off *req_len* bytes from head or tail of the (valid) data area. If *req_len* is greater than zero, front bytes are being shaved off, if it's smaller, from the back (and if it is zero, the mbuf will stay bearded). This function does not move data in any way, but is used to manipulate the data area pointer and data length variable of the mbuf in a non-clobbering way.

m_apply(*struct mbuf *m*, *int off*, *int len*, *int (*f)(void *, void *, unsigned int)*, *void *arg*)
Apply function *f* to the data in an mbuf chain starting *off* bytes from the beginning, continuing for *len* bytes. Neither *off* nor *len* may be negative. *arg* will be supplied as first argument for *f*, the second argument will be the pointer to the data buffer of a packet (starting after *off* bytes in the stream), and the third argument is the amount of data in bytes in this call. If *f* returns something not equal to zero **m_apply**() will bail out, returning the return code of *f*. Upon successful completion it will return zero.

m_free(*struct mbuf *m*)
Frees mbuf *m*.

m_freem(*struct mbuf *m*)
Frees the mbuf chain beginning with *m*. This function contains the elementary sanity check for a `NULL` pointer.

mtod(*struct mbuf *m*, *datatype*)
Returns a pointer to the data contained in the specified mbuf *m*, type-casted to the specified data type *datatype*. Implemented as a macro.

MGET(*struct mbuf *m*, *int how*, *int type*)
Allocates mbuf *m* and initializes it to contain internal data. See **m_get**(). Implemented as a macro.

MGETHDR(*struct mbuf *m*, *int how*, *int type*)
Allocates mbuf *m* and initializes it to contain a packet header. See **m_gethdr**(). Implemented as a macro.

MEXTMALLOC(*struct mbuf *m*, *int len*, *int how*)
Allocates external storage of size *len* for mbuf *m*. The *how* parameter is a choice of `M_WAIT / M_DONTWAIT` from caller. The flag `M_EXT` is set upon success. Implemented as a macro.

MEXTADD(*struct mbuf *m*, *void *buf*, *int size*, *int type*, *void (*free)(struct mbuf *, void *, size_t, void *)*, *void *arg*)
Adds pre-allocated external storage *buf* to a normal mbuf *m*; the parameters *size*, *type*, *free* and *arg* describe the external storage. *size* is the size of the storage, *type* describes its malloc(9) type, *free* is a free routine (if not the usual one), and *arg* is a possible argument to the free routine. The flag M_EXT is set upon success. Implemented as a macro. If a free routine is specified, it will be called when the mbuf is freed. In the case of former, the first argument for a free routine is the mbuf *m* and the routine is expected to free it in addition to the external storage pointed by second argument. In the case of latter, the first argument for the routine is NULL.

MCLGET(*struct mbuf *m*, *int how*)
Allocates and adds an mbuf cluster to a normal mbuf *m*. The *how* parameter is a choice of M_WAIT / M_DONTWAIT from caller. The flag M_EXT is set upon success. Implemented as a macro.

M_COPY_PKTHDR(*struct mbuf *to*, *struct mbuf *from*)
Copies the mbuf pkthdr from mbuf *from* to mbuf *to*. *from* must have the type flag M_PKTHDR set, and *to* must be empty. Implemented as a macro.

M_MOVE_PKTHDR(*struct mbuf *to*, *struct mbuf *from*)
Moves the mbuf pkthdr from mbuf *from* to mbuf *to*. *from* must have the type flag M_PKTHDR set, and *to* must be empty. The flag M_PKTHDR in mbuf *from* will be cleared.

M_ALIGN(*struct mbuf *m*, *int len*)
Sets the data pointer of a newly allocated mbuf *m* to *len* bytes from the end of the mbuf data area, so that *len* bytes of data written to the mbuf *m*, starting at the data pointer, will be aligned to the end of the data area. Implemented as a macro.

MH_ALIGN(*struct mbuf *m*, *int len*)
Sets the data pointer of a newly allocated packetheader mbuf *m* to *len* bytes from the end of the mbuf data area, so that *len* bytes of data written to the mbuf *m*, starting at the data pointer, will be aligned to the end of the data area. Implemented as a macro.

M_LEADINGSPACE(*struct mbuf *m*)
Returns the amount of space available before the current start of valid data in mbuf *m*. Returns 0 if the mbuf data part is shared across multiple mbufs (i.e. not writable). Implemented as a macro.

M_TRAILINGSPACE(*struct mbuf *m*)
Returns the amount of space available after the current end of valid data in mbuf *m*. Returns 0 if the mbuf data part is shared across multiple mbufs (i.e. not writable). Implemented as a macro.

M_PREPEND(*struct mbuf *m*, *int plen*, *int how*)
Prepends space of size *plen* to mbuf *m*. If a new mbuf must be allocated, *how* specifies whether to wait. If *how* is M_DONTWAIT and allocation fails, the original mbuf chain is freed and *m* is set to NULL. Implemented as a macro.

MCHTYPE(*struct mbuf *m*, *int type*)
Change mbuf *m* to new type *type*. Implemented as a macro.

MFREE(*struct mbuf *m*, *struct mbuf *n*)
Frees a single mbuf *m* and places the successor, if any, in mbuf *n*. Implemented as a macro.

FILES

The **mbuf** management functions are implemented within the file sys/kern/uipc_mbuf.c. Function prototypes, and the functions implemented as macros are located in sys/sys/mbuf.h. Both pathnames are relative to the root of the NetBSD source tree, /usr/src.

SEE ALSO

`/usr/share/doc/smm/18.net`, `netstat`(1), `m_tag`(9), `malloc`(9)

Jun-ichiro Hagino, "Mbuf issues in 4.4BSD IPv6/IPsec support (experiences from KAME IPv6/IPsec implementation)", *Proceedings of the freenix track: 2000 USENIX annual technical conference*, June 2000.

AUTHORS

The original mbuf data structures were designed by Rob Gurwitz when he did the initial TCP/IP implementation at BBN.

Further extensions and enhancements were made by Bill Joy, Sam Leffler, and Mike Karels at CSRG.

Current implementation of external storage by Matt Thomas ⟨matt@3am-software.com⟩ and Jason R. Thorpe ⟨thorpej@NetBSD.org⟩.

NAME

MCA, **mca_intr_establish**, **mca_intr_disestablish**, **mca_intr_evcnt**, **mca_conf_read**, **mca_conf_write** — MicroChannel Architecture bus

SYNOPSIS

```
#include <machine/bus.h>
#include <dev/mca/mcavar.h>
#include <dev/mca/mcadevs.h>

void *
mca_intr_establish(mca_chipset_tag_t mc, mca_intr_handle_t hdl, int level,
    int (*handler)(void *), void *arg);

void
mca_intr_disestablish(mca_chipset_tag_t mc, mca_intr_handle_t hdl);

const struct evcnt *
mca_intr_evcnt(mca_chipset_tag_t mc, mca_intr_handle_t hdl);

int
mca_conf_read(mca_chipset_tag_t mc, int slot, int reg);

void
mca_conf_write(mca_chipset_tag_t mc, int slot, int reg, int data);
```

DESCRIPTION

The **MCA** device provides support for IBM's MicroChannel Architecture bus found on IBM PS/2 systems and selected workstations. It was designed as a replacement bus for the ISA bus found on IBM's older machines. However, the bus specifications were only available under license, so MCA did not achieve widespread acceptance in the industry.

Being a replacement for the ISA bus, the MCA bus does share some similar aspects with the ISA bus. Some MCA devices can be detected via the usual ISA-style probing. However, most device detection is done through the Programmable Option Select (POS) registers. These registers provide a window into a device to determine device-specific properties and configuration. The configuration of devices and their POS registers is performed using IBM's system configuration software.

The MCA bus uses level-triggered interrupts while the ISA bus uses edge-triggered interrupts. Level triggered interrupts have the advantage that they can be shared among multiple device. Therefore, most MCA-specific devices should be coded with shared interrupts in mind.

DATA TYPES

Drivers for devices attached to the MCA bus will make use of the following data types:

mca_chipset_tag_t
: Chipset tag for the MCA bus.

mca_intr_handle_t
: The opaque handle describing an established interrupt handler.

struct mca_attach_args
: A structure use to inform the driver of MCA bus properties. It contains the following members:

```
bus_space_tag_t ma_iot;              /* MCA I/O space tag */
bus_space_tag_t ma_memt;      /* MCA mem space tag */
bus_dma_tag_t ma_dmat;        /* MCA DMA tag */
int ma_slot;                  /* MCA slot number */
```

```
int ma_pos[8];                  /* MCA POS values */
int ma_id;                      /* MCA device */
```

FUNCTIONS

mca_intr_establish(*mc*, *hdl*, *level*, *handler*, *arg*)
Establish a MCA interrupt handler on the MCA bus specified by *mc* for the interrupt described completely by *hdl*. The priority of the interrupt is specified by *level*. When the interrupt occurs the function *handler* is called with argument *arg*.

mca_intr_disestablish(*mc*, *hdl*)
Dis-establish the interrupt handler on the MCA bus specified by *mc* for the interrupt described completely *hdl*.

mca_intr_evcnt(*mc*, *hdl*)
Do interrupt event counting on the MCA bus specified by *mc* for the event described completely by *hdl*.

mca_conf_read(*mc*, *slot*, *reg*)
Read the POS register *reg* for the device in slot *slot* on the MCA bus specified by *mc*.

mca_conf_write(*mc*, *slot*, *reg*, *data*)
Write data *data* to the POS register *reg* for the device in slot *slot* on the MCA bus specified by *mc*.

AUTOCONFIGURATION

The MCA bus is a direct-connection bus. During autoconfiguration, the parent specifies the MCA device ID for the found device in the *ma_id* member of the *mca_attach_args* structure. Drivers should match on the device ID. Device capabilities and configuration information should be read from device POS registers using **mca_conf_read**(). Some important configuration information found in the POS registers include the I/O base address, memory base address and interrupt number. The location of these configurable options with the POS registers are device specific.

DMA SUPPORT

The MCA bus supports 32-bit, bidirectional DMA transfers. Currently, no machine-independent support for MCA DMA is available.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent MCA subsystem can be found. All pathnames are relative to `/usr/src`.

The MCA subsystem itself is implemented within the file `sys/dev/mca/mca_subr.c`. Machine-dependent portions can be found in `sys/arch/<arch>/mca/mca_machdep.c`. The database of known devices exists within the file *sys/dev/mca/mcadevs_data.h* and is generated automatically from the file `sys/dev/mca/mcadevs`. New vendor and product identifiers should be added to this file. The database can be regenerated using the Makefile `sys/dev/mca/Makefile.mcadevs`.

A good source of information about MCA devices is IBM's system configuration disk. The disk contains .adf files which describe the location of device configuration options in the POS registers.

SEE ALSO

`mca`(4), `autoconf`(9), `bus_dma`(9), `bus_space`(9), `driver`(9), `isa`(9)

BUGS

The machine-independent **MCA** driver does not currently support DMA. MCA devices which require DMA operation currently access the DMA capabilities directly.

NAME

memcmp — compare byte string

SYNOPSIS

```
#include <sys/systm.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.

STANDARDS

The **memcmp**() function conforms to ANSI X3.159-1989 ("ANSI C89").

NAME

memcpy — copy byte string

SYNOPSIS

```
#include <systm.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(9).

RETURN VALUES

The **memcpy**() function returns the original value of *dst*.

SEE ALSO

memmove(9)

STANDARDS

The **memcpy**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

memmove — copy byte string

SYNOPSIS

```
#include <sys/systm.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

memcpy(9)

STANDARDS

The **memmove**() function conforms to ANSI X3.159-1989 ("ANSI C89").

NAME

memoryallocators — introduction to kernel memory allocators

DESCRIPTION

The NetBSD kernel provides several memory allocators, each with different characteristics and purpose. This document summarizes the main differences between them.

The Malloc Allocator

The malloc(9) allocator can be used for variable-sized allocations in the kernel address space. It is interrupt-safe, requires no setup (see below), and is considered to be stable (given the number of years it has been in the kernel).

This interface also allows associating a "type" with an allocation to indicate what subsystem is using the memory allocated, thus providing statistics as to the memory usage of different kernel subsystems. To define a type, one should use the MALLOC_DEFINE macro, otherwise, one of the built-in types, like M_TEMP can be used.

See malloc(9) for more details.

The Kmem Allocator

The kmem allocator is modelled after an interface of similar name implemented in Solaris, and is under active development.

It is implemented on-top of the vmem(9) resource allocator (beyond the scope of this document), meaning it will be using pool_cache(9) internally to speed-up common (small) sized allocations.

Like malloc(9), it requires no setup, but can't be used yet from interrupt context.

See kmem_alloc(9), kmem_zalloc(9), and kmem_free(9) for more details.

The Pool Allocator

The pool(9) allocator is a fixed-size memory allocator. It requires setup (to initialize a memory pool) and is interrupt-safe.

See pool(9) for more details.

The Pool Cache Allocator

The pool cache allocator works on-top of the pool(9) allocator, also allowing fixed-size allocation only, requires setup, and is interrupt-safe.

The pool cache allocator is expected to be faster than other allocators, including the "normal" pool allocator.

In the future this allocator is expected to have a per-CPU cache.

See pool_cache(9) for more details.

The UVM Kernel Memory Allocator

This is a low-level memory allocator interface. It allows variable-sized allocations in multiples of PAGE_SIZE, and can be used to allocate both wired and pageable kernel memory.

See uvm(9) for more details.

SEE ALSO

free(9), intro(9), kmem_alloc(9), kmem_free(9), kmem_zalloc(9), malloc(9), pool(9), pool_cache(9), uvm(9), vmem(9)

AUTHORS

Elad Efrat ⟨elad@NetBSD.org⟩
YAMAMOTO Takashi ⟨yamt@NetBSD.org⟩

NAME

memset — write a byte to byte string

SYNOPSIS

```
#include <sys/systm.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*.

STANDARDS

The **memset**() function conforms to ANSI X3.159-1989 ("ANSI C89").

NAME

mi_switch — machine independent context switch prelude

SYNOPSIS

int
mi_switch(*struct lwp *l*);

DESCRIPTION

The **mi_switch**() function implements the machine-independent prelude to an LWP context switch. It is called from only a few distinguished places in the kernel code as a result of the principle of non-preemptable kernel mode execution. The three major uses of **mi_switch**() can be enumerated as follows:

1. From within cv_wait(9) and associated methods when the current LWP voluntarily relinquishes the CPU to wait for some resource to become available.
2. From within preempt(9) when the current LWP voluntarily relinquishes the CPU or when the kernel prepares a return to user-mode execution.
3. In the signal handling code if a signal is delivered that causes an LWP to stop (see issignal(9)).

mi_switch() records the amount of time the current LWP has been running in the LWP structure and checks this value against the CPU time limits allocated to the LWP (see getrlimit(2)). Exceeding the soft limit results in a SIGXCPU signal to be posted to the LWP, while exceeding the hard limit will cause a SIGKILL.

Unless *l->l_switchto* is not NULL, **mi_switch**() will call **sched_nextlwp**() to select a new LWP from the scheduler's runqueue structures. If no runnable LWP is found, the idle LWP is used. If the new LWP is not equal to the current one, **mi_switch**() will hand over control to the machine-dependent function cpu_switchto(9) to switch to the new LWP.

mi_switch() has to be called with the LWP lock held (through calling **lwp_lock**() first) and at the splsched(9) interrupt protection level. It returns with the LWP lock released.

RETURN VALUES

mi_switch() returns 1 if a context switch was performed to a different LWP, 0 otherwise.

SEE ALSO

condvar(9), cpu_switchto(9), csf(9), pmap(9), ras(9), sched_4bsd(9), splsched(9)

NAME

microseq — ppbus microseqencer developer's guide

SYNOPSIS

```
#include <sys/types.h>
#include <dev/ppbus/ppbus_conf.h>
#include <dev/ppbus/ppbus_msq.h>
```

DESCRIPTION

See ppbus(4) for **ppbus** description and general info about the microsequencer.

The purpose of this document is to encourage developers to use the microsequencer mechanism in order to have:

1. a uniform programming model
2. efficient code

Before using microsequences, you are encouraged to look at the atppc(4) microsequencer implementation and an example of how using it in vpo(4).

PPBUS register model

Background

The parallel port model chosen for ppbus(4) is the PC parallel port model. Thus, any register described later has the same semantic than its counterpart in a PC parallel port. For more info about ISA/ECP programming, get the Microsoft standard referenced "Extended Capabilities Port Protocol and ISA interface Standard". Registers described later are standard parallel port registers.

Mask macros are defined in the standard ppbus(4) include files for each valid bit of parallel port registers.

Data register

In compatible or nibble mode, writing to this register will drive data to the parallel port data lines. In any other mode, drivers may be tri-stated by setting the direction bit (PCD) in the control register. Reads to this register return the value on the data lines.

Device status register

This read-only register reflects the inputs on the parallel port interface.

Bit	*Name*	*Description*
7	nBUSY	inverted version of parallel port Busy signal
6	nACK	version of parallel port nAck signal
5	PERROR	version of parallel port PERROR signal
4	SELECT	version of parallel port Select signal
3	nFAULT	version of parallel port nFault signal

Others are reserved and return undefined result when read.

Device control register

This register directly controls several output signals as well as enabling some functions.

Bit	*Name*	*Description*
5	PCD	direction bit in extended modes
4	IRQENABLE	1 enables an interrupt on the rising edge of nAck

3	SELECTIN	inverted and driven as parallel port nSelectin signal
2	nINIT	driven as parallel port nInit signal
1	AUTOFEED	inverted and driven as parallel port nAutoFd signal
0	STROBE	inverted and driven as parallel port nStrobe signal

MICROINSTRUCTIONS

Description

Microinstructions are either parallel port accesses, program iterations, submicrosequence or C calls. The parallel port must be considered as the logical model described in ppbus(4).

Available microinstructions are:

```
#define MS_OP_GET       0     /* get <ptr>, <len>                  */
#define MS_OP_PUT       1     /* put <ptr>, <len>                  */
#define MS_OP_RFETCH    2     /* rfetch <reg>, <mask>, <ptr>       */
#define MS_OP_RSET      3     /* rset <reg>, <mask>, <mask>        */
#define MS_OP_RASSERT   4     /* rassert <reg>, <mask>             */
#define MS_OP_DELAY     5     /* delay <val>                       */
#define MS_OP_SET       6     /* set <val>                         */
#define MS_OP_DBRA      7     /* dbra <offset>                     */
#define MS_OP_BRSET     8     /* brset <mask>, <offset>            */
#define MS_OP_BRCLEAR   9     /* brclear <mask>, <offset>          */
#define MS_OP_RET       10    /* ret <retcode>                     */
#define MS_OP_C_CALL    11    /* c_call <function>, <parameter>    */
#define MS_OP_PTR       12    /* ptr <pointer>                     */
#define MS_OP_ADELAY    13    /* adelay <val>                      */
#define MS_OP_BRSTAT    14    /* brstat <mask>, <mask>, <offset>   */
#define MS_OP_SUBRET    15    /* subret <code>                     */
#define MS_OP_CALL      16    /* call <microsequence>                    */
#define MS_OP_RASSERT_P       17     /* rassert_p <iter>, <reg>          */
#define MS_OP_RFETCH_P  18    /* rfetch_p <iter>, <reg>, <mask>    */
#define MS_OP_TRIG      19    /* trigger <reg>, <len>, <array>     */
```

Execution context

The *execution context* of microinstructions is:

- the *program counter* which points to the next microinstruction to execute either in the main microsequence or in a subcall
- the current value of *ptr* which points to the next char to send/receive
- the current value of the internal *branch register*

This data is modified by some of the microinstructions, not all.

MS_OP_GET and MS_OP_PUT

are microinstructions used to do either predefined standard IEEE1284-1994 transfers or programmed non-standard I/O.

MS_OP_RFETCH - Register FETCH

is used to retrieve the current value of a parallel port register, apply a mask and save it in a buffer.

Parameters:

1. register
2. character mask
3. pointer to the buffer

Predefined macro: MS_RFETCH(reg,mask,ptr)

MS_OP_RSET - Register SET
is used to assert/clear some bits of a particular parallel port register, two masks are applied.

Parameters:

1. register
2. mask of bits to assert
3. mask of bits to clear

Predefined macro: MS_RSET(reg,assert,clear)

MS_OP_RASSERT - Register ASSERT
is used to assert all bits of a particular parallel port register.

Parameters:

1. register
2. byte to assert

Predefined macro: MS_RASSERT(reg,byte)

MS_OP_DELAY - microsecond DELAY
is used to delay the execution of the microsequence.

Parameter:

1. delay in microseconds

Predefined macro: MS_DELAY(delay)

MS_OP_SET - SET internal branch register
is used to set the value of the internal branch register.

Parameter:

1. integer value

Predefined macro: MS_SET(accum)

MS_OP_DBRA - &Do BRAnch
is used to branch if internal branch register decremented by one result value is positive.

Parameter:

1. integer offset in the current executed (sub)microsequence. Offset is added to the index of the next microinstruction to execute.

Predefined macro: MS_DBRA(offset)

MS_OP_BRSET - BRanch on SET
is used to branch if some of the status register bits of the parallel port are set.

Parameter:

1. bits of the status register
2. integer offset in the current executed (sub)microsequence. Offset is added to the index of the next microinstruction to execute.

Predefined macro: MS_BRSET(mask,offset)

MS_OP_BRCLEAR - BRanch on CLEAR
is used to branch if some of the status register bits of the parallel port are cleared.

Parameter:

1. bits of the status register
2. integer offset in the current executed (sub)microsequence. Offset is added to the index of the next microinstruction to execute.

Predefined macro: MS_BRCLEAR(mask,offset)

MS_OP_RET - RETurn
is used to return from a microsequence. This instruction is mandatory. This is the only way for the microsequencer to detect the end of the microsequence. The return code is returned in the integer pointed by the (int *) parameter of the ppb_MS_microseq().

Parameter:

1. integer return code

Predefined macro: MS_RET(code)

MS_OP_C_CALL - C function CALL
is used to call C functions from microsequence execution. This may be useful when a non-standard I/O is performed to retrieve a data character from the parallel port.

Parameter:

1. the C function to call
2. the parameter to pass to the function call

The C function shall be declared as a `int(*)(void *p, char *ptr)`. The ptr parameter is the current position in the buffer currently scanned.

Predefined macro: MS_C_CALL(func,param)

MS_OP_PTR - initialize internal PTR
is used to initialize the internal pointer to the currently scanned buffer. This pointer is passed to any C call (see above).

Parameter:

1. pointer to the buffer that shall be accessed by **xxx_P**() microsequence calls. Note that this pointer is automatically incremented during **xxx_P**() calls.

Predefined macro: MS_PTR(ptr)

MS_OP_ADELAY - do an Asynchronous DELAY
is used to make a `cv_timedwait`(9) during microsequence execution.

Parameter:

1. delay in ms

Predefined macro: MS_ADELAY(delay)

MS_OP_BRSTAT - BRanch on STATe
is used to branch on status register state condition.

Parameter:

1. mask of asserted bits. Bits that shall be asserted in the status register are set in the mask
2. mask of cleared bits. Bits that shall be cleared in the status register are set in the mask
3. integer offset in the current executed (sub)microsequence. Offset is added to the index of the next microinstruction to execute.

Predefined macro: MS_BRSTAT(asserted_bits,clear_bits,offset)

MS_OP_SUBRET - SUBmicrosequence RETurn
is used to return from the submicrosequence call. This action is mandatory before a RET call. Some microinstructions (PUT, GET) may not be callable within a submicrosequence.

No parameter.

Predefined macro: MS_SUBRET()

MS_OP_CALL - submicrosequence CALL
is used to call a submicrosequence. A submicrosequence is a microsequence with a SUBRET call. Parameter:

1. the submicrosequence to execute

Predefined macro: MS_CALL(microseq)

MS_OP_RASSERT_P - Register ASSERT from internal PTR
is used to assert a register with data currently pointed by the internal PTR pointer. Parameter:

1. amount of data to write to the register
2. register

Predefined macro: MS_RASSERT_P(iter,reg)

MS_OP_RFETCH_P - Register FETCH to internal PTR
is used to fetch data from a register. Data is stored in the buffer currently pointed by the internal PTR pointer. Parameter:

1. amount of data to read from the register
2. register
3. mask applied to fetched data

Predefined macro: MS_RFETCH_P(iter,reg,mask)

MS_OP_TRIG - TRIG register

is used to trigger the parallel port. This microinstruction is intended to provide a very efficient control of the parallel port. Triggering a register is writing data, wait a while, write data, wait a while... This allows to write magic sequences to the port. Parameter:

1. amount of data to read from the register
2. register
3. size of the array
4. array of unsigned chars. Each couple of u_chars define the data to write to the register and the delay in us to wait. The delay is limited to 255 us to simplify and reduce the size of the array.

Predefined macro: MS_TRIG(reg,len,array)

MICROSEQUENCES

C structures

```
union ppb_insarg {
     int     i;
     char    c;
     void    *p;
     int     (* f)(void *, char *);
};

struct ppb_microseq {
     int                    opcode;         /* microins. opcode */
     union ppb_insarg       arg[PPB_MS_MAXARGS];    /* arguments */
};
```

Using microsequences

To instantiate a microsequence, just declare an array of ppb_microseq structures and initialize it as needed. You may either use predefined macros or code directly your microinstructions according to the ppb_microseq definition. For example,

```
     struct ppb_microseq select_microseq[] = {

             /* parameter list
              */
             #define SELECT_TARGET    MS_PARAM(0, 1, MS_TYP_INT)
             #define SELECT_INITIATOR MS_PARAM(3, 1, MS_TYP_INT)

             /* send the select command to the drive */
             MS_DASS(MS_UNKNOWN),
             MS_CASS(H_nAUTO | H_nSELIN |  H_INIT | H_STROBE),
             MS_CASS( H_AUTO | H_nSELIN |  H_INIT | H_STROBE),
             MS_DASS(MS_UNKNOWN),
             MS_CASS( H_AUTO | H_nSELIN | H_nINIT | H_STROBE),

             /* now, wait until the drive is ready */
             MS_SET(VP0_SELTMO),
/* loop: */     MS_BRSET(H_ACK, 2 /* ready */),
             MS_DBRA(-2 /* loop */),
/* error: */    MS_RET(1),
/* ready: */    MS_RET(0)
```

```
};
```

Here, some parameters are undefined and must be filled before executing the microsequence. In order to initialize each microsequence, one should use the **ppb_MS_init_msq**() function like this:

```
ppb_MS_init_msq(select_microseq, 2,
                SELECT_TARGET, 1 << target,
                SELECT_INITIATOR, 1 << initiator);
```

and then execute the microsequence.

The microsequencer

The microsequencer is executed either at ppbus or adapter level (see ppbus(4) for info about ppbus system layers). Most of the microsequencer is executed at atppc(4) level to avoid ppbus(4) to adapter function call overhead. But some actions like deciding whereas the transfer is IEEE1284-1994 compliant are executed at ppbus(4) layer.

SEE ALSO

atppc(4), ppbus(4), vpo(4)

HISTORY

The **microseq** manual page first appeared in FreeBSD 3.0.

AUTHORS

This manual page is based on the FreeBSD **microseq** manual page and was update for the NetBSD port by Gary Thorpe.

BUGS

Only one level of submicrosequences is allowed.

When triggering the port, maximum delay allowed is 255 us.

NAME

microtime — realtime system clock

SYNOPSIS

```
#include <sys/time.h>

void
microtime(struct timeval *tv);
```

DESCRIPTION

microtime() returns the current value of the system realtime clock in the structure pointed to by the argument *tv*. The system realtime clock is guaranteed to be monotonically increasing at all times. As such, all calls to **microtime**() are guaranteed to return a system time greater than or equal to the system time returned in any previous calls.

SEE ALSO

settimeofday(2), hardclock(9), hz(9), inittodr(9), time_second(9)

CODE REFERENCES

The implementation of the **microtime**() function is machine dependent, hence its location in the source code tree varies from architecture to architecture.

BUGS

Despite the guarantee that the system realtime clock will always be monotonically increasing, it is always possible for the system clock to be manually reset by the system administrator to any date.

NAME

binuptime, **getbinuptime**, **microuptime**, **getmicrouptime**, **nanouptime**, **getnanouptime** — get the time elapsed since boot

SYNOPSIS

```
#include <sys/time.h>

void
binuptime(struct bintime *bt);

void
getbinuptime(struct bintime *bt);

void
microuptime(struct timeval *tv);

void
getmicrouptime(struct timeval *tv);

void
nanouptime(struct timespec *ts);

void
getnanouptime(struct timespec *tsp);
```

DESCRIPTION

The **binuptime**() and **getbinuptime**() functions store the time elapsed since boot as a *struct bintime* at the address specified by *bt*. The **microuptime**() and **getmicrouptime**() functions perform the same utility, but record the elapsed time as a *struct timeval* instead. Similarly the **nanouptime**() and **getnanouptime**() functions store the elapsed time as a *struct timespec*.

The **binuptime**(), **microuptime**(), and **nanouptime**() functions always query the timecounter to return the current time as precisely as possible. Whereas **getbinuptime**(), **getmicrouptime**(), and **getnanouptime**() functions are abstractions which return a less precise, but faster to obtain, time.

The intent of the **getbinuptime**(), **getmicrouptime**(), and **getnanouptime**() functions is to enforce the user's preference for timer accuracy versus execution time.

SEE ALSO

bintime(9), getbintime(9), getmicrotime(9), getnanotime(9), microtime(9), nanotime(9), tvtohz(9)

AUTHORS

This manual page was written by Kelly Yancey ⟨kbyanc@posi.net⟩.

NAME

mstohz — convert milliseconds to system clock ticks

SYNOPSIS

```
#include <sys/param.h>

int
mstohz(int ms);
```

DESCRIPTION

mstohz can be used to convert time in milliseconds to system clock ticks, as used by the callout(9) facility, in an integer-overflow safe way.

This function is implemented as a define in the <*sys/param.h*> header. Individual ports can have a processor-specific, more efficient version implemented in their <*machine/param.h*> header as a define.

RETURN VALUES

The return value is the number of clock ticks for the specified value.

SEE ALSO

callout(9)

BUGS

The machine-independent **mstohz**() function does not make use of expensive 64-bit integer arithmetic, so the result will be rounded down to one second if the parameter is larger than 131072 milliseconds.

NAME

mutex, **mutex_init**, **mutex_destroy**, **mutex_enter**, **mutex_exit**, **mutex_owned**, **mutex_spin_enter**, **mutex_spin_exit**, **mutex_tryenter** — mutual exclusion primitives

SYNOPSIS

```
#include <sys/mutex.h>

void
mutex_init(kmutex_t *mtx, kmutex_type_t type, int ipl);

void
mutex_destroy(kmutex_t *mtx);

void
mutex_enter(kmutex_t *mtx);

void
mutex_exit(kmutex_t *mtx);

int
mutex_owned(kmutex_t *mtx);

void
mutex_spin_enter(kmutex_t *mtx);

void
mutex_spin_exit(kmutex_t *mtx);

int
mutex_tryenter(kmutex_t *mtx);

options DIAGNOSTIC
options LOCKDEBUG
```

DESCRIPTION

Mutexes are used in the kernel to implement mutual exclusion among LWPs (lightweight processes) and interrupt handlers.

The *kmutex_t* type provides storage for the mutex object. This should be treated as an opaque object and not examined directly by consumers.

Mutexes replace the spl(9) system traditionally used to provide synchronization between interrupt handlers and LWPs, and in combination with reader / writer locks replace the lockmgr(9) facility.

OPTIONS

options DIAGNOSTIC

Kernels compiled with the DIAGNOSTIC option perform basic sanity checks on mutex operations.

options LOCKDEBUG

Kernels compiled with the LOCKDEBUG option perform potentially CPU intensive sanity checks on mutex operations.

FUNCTIONS

mutex_init(*mtx*, *type*, *ipl*)

Dynamically initialize a mutex for use.

No other operations can be performed on a mutex until it has been initialized. Once initialized, all types of mutex are manipulated using the same interface. Note that **mutex_init**() may block in order to allocate memory.

The *type* argument must be given as MUTEX_DEFAULT. Other constants are defined but are for low-level system use and are not an endorsed, stable part of the interface.

The type of mutex returned depends on the *ipl* argument:

IPL_NONE, or one of the IPL_SOFT∗ constants

An adaptive mutex will be returned. Adaptive mutexes provide mutual exclusion between LWPs, and between LWPs and soft interrupt handlers.

Adaptive mutexes cannot be acquired from a hardware interrupt handler. An LWP may either sleep or busy-wait when attempting to acquire an adaptive mutex that is already held.

IPL_VM, IPL_SCHED, IPL_HIGH

A spin mutex will be returned. Spin mutexes provide mutual exclusion between LWPs, and between LWPs and interrupt handlers.

The *ipl* argument is used to pass a system interrupt priority level (IPL) that will block all interrupt handlers that may try to acquire the mutex.

LWPs that own spin mutexes may not sleep, and therefore must not try to acquire adaptive mutexes or other sleep locks.

A processor will always busy-wait when attempting to acquire a spin mutex that is already held.

See spl(9) for further information on interrupt priority levels (IPLs).

mutex_destroy(*mtx*)

Release resources used by a mutex. The mutex may not be used after it has been destroyed. **mutex_destroy**() may block in order to free memory.

mutex_enter(*mtx*)

Acquire a mutex. If the mutex is already held, the caller will block and not return until the mutex is acquired.

Mutexes and other types of locks must always be acquired in a consistent order with respect to each other. Otherwise, the potential for system deadlock exists.

Adaptive mutexes and other types of lock that can sleep may not be acquired while a spin mutex is held by the caller.

mutex_exit(*mtx*)

Release a mutex. The mutex must have been previously acquired by the caller. Mutexes may be released out of order as needed.

mutex_owned(*mtx*)

For adaptive mutexes, return non-zero if the current LWP holds the mutex. For spin mutexes, return non-zero if the mutex is held, potentially by the current processor. Otherwise, return zero.

mutex_owned() is provided for making diagnostic checks to verify that a lock is held. For example:

```
KASSERT(mutex_owned(&driver_lock));
```

It should not be used to make locking decisions at run time, or to verify that a lock is unheld.

mutex_spin_enter(*mtx*)

Equivalent to **mutex_enter**(), but may only be used when it is known that *mtx* is a spin mutex. On some architectures, this can substantially reduce the cost of acquring a spin mutex.

mutex_spin_exit(*mtx*)

Equivalent to **mutex_exit**(), but may only be used when it is known that *mtx* is a spin mutex. On some architectures, this can substantially reduce the cost of releasing an unheld spin mutex.

mutex_tryenter(*mtx*)

Try to acquire a mutex, but do not block if the mutex is already held. Returns non-zero if the mutex was acquired, or zero if the mutex was already held.

mutex_tryenter() can be used as an optimization when acquiring locks in the the wrong order. For example, in a setting where the convention is that `first_lock` must be acquired before `second_lock`, the following can be used to optimistically lock in reverse order:

```
/* We hold second_lock, but not first_lock. */
KASSERT(mutex_owned(&second_lock));

if (!mutex_tryenter(&first_lock)) {
        /* Failed to get it - lock in the correct order. */
        mutex_exit(&second_lock);
        mutex_enter(&first_lock);
        mutex_enter(&second_lock);

        /*
         * We may need to recheck any conditions the code
         * path depends on, as we released second_lock
         * briefly.
         */
}
```

CODE REFERENCES

This section describes places within the NetBSD source tree where code implementing mutexes can be found. All pathnames are relative to `/usr/src`.

The core of the mutex implementation is in `sys/kern/kern_mutex.c`.

The header file `sys/sys/mutex.h` describes the public interface, and interfaces that machine-dependent code must provide to support mutexes.

SEE ALSO

`condvar`(9), `mb`(9), `rwlock`(9), `spl`(9)

Jim Mauro and Richard McDougall, *Solaris Internals: Core Kernel Architecture,*, *Prentice Hall*, 2001, ISBN 0-13-022496-0.

HISTORY

The mutex primitives first appeared in NetBSD 5.0.

NAME

namecache, **cache_lookup**, **cache_revlookup**, **cache_enter**, **cache_purge**, **cache_purgevfs**, **namecache_print** — name lookup cache

SYNOPSIS

```
#include <sys/namei.h>
#include <sys/proc.h>
#include <sys/uio.h>
#include <sys/vnode.h>

int
cache_lookup(struct vnode *dvp, struct vnode **vpp,
    struct componentname *cnp);

int
cache_revlookup(struct vnode *vp, struct vnode *dvp, char **bpp, char *bufp);

void
cache_enter(struct vnode *dvp, struct vnode *vp, struct componentname *cnp);

void
cache_purge(struct vnode *vp);

void
cache_purgevfs(struct mount *mp);

void
namecache_print(struct vnode *vp, void (*func)(const char *, ...));
```

DESCRIPTION

The name lookup cache is the mechanism to allow the file system type dependent algorithms to quickly resolve a file's vnode from its pathname. The name lookup cache is generally accessed through the higher-level namei(9) interface.

The name of the file is used to lookup an entry associated with the file in the name lookup cache. If no entry is found, one is created for it. If an entry is found, the information obtained from the cache lookup contains information about the file which is useful to the file system type dependent functions.

The name lookup cache is managed by a least recently used (LRU) algorithm so frequently used names will hang around. The cache itself is a hash table called *nchashtbl*, containing *namecache* entries that are allocated dynamically from a kernel memory pool. Each entry has the following structure:

```
#define NCHNAMLEN     31     /* maximum name segment length */
struct  namecache {
        LIST_ENTRY(namecache) nc_hash;  /* hash chain */
        TAILQ_ENTRY(namecache) nc_lru;  /* LRU chain */
        LIST_ENTRY(namecache) nc_vhash; /* directory hash chain */
        LIST_ENTRY(namecache) nc_dvlist;
        struct  vnode *nc_dvp;          /* vnode of parent of name */
        LIST_ENTRY(namecache) nc_vlist;
        struct  vnode *nc_vp;           /* vnode the name refers to */
        int     nc_flags;               /* copy of componentname's ISWHITEOUT */
        char    nc_nlen;                /* length of name */
        char    nc_name[NCHNAMLEN];     /* segment name */
};
```

For simplicity (and economy of storage), names longer than a maximum length of NCHNAMLEN are not cached; they occur infrequently in any case, and are almost never of interest.

Each *namecache* entry can appear on two hash chains in addition to *nshashtbl*: *ncvhashtbl* (the name cache directory hash chain), and *nclruhead* (the name cache LRU chain). The hash chains are indexed by a hash value obtained from the file's name and the address of its parent directory vnode.

Functions which access to the name cache pass arguments in the partially initialised *componentname* structure. See vnodeops(9) for details on this structure.

FUNCTIONS

cache_lookup(*dvp*, *vpp*, *cnp*)

Look for a name in the cache. **cache_lookup**() is called with *dvp* pointing to the vnode of the directory to search and *cnp* pointing to the partially initialised component structure. *cnp->cn_nameptr* points to the name of the entry being sought, *cnp->cn_namelen* tells the length of the name, and *cnp->cn_hash* contains a hash of the name. If the lookup succeeds, the vnode is locked, stored in *vpp* and a status of zero is returned. If the locking fails for whatever reason, the vnode is unlocked and the error is returned. If the lookup determines that the name does not exist any longer, a status of ENOENT is returned. If the lookup fails, a status of -1 is returned.

cache_revlookup(*vp*, *dvp*, *bpp*, *bufp*)

Scan cache looking for name of directory entry pointing at *vp* and fill in *dvpp*. If *bufp* is non-NULL, also place the name in the buffer which starts at *bufp*, immediately before *bpp*, and move bpp backwards to point at the start of it. Returns 0 on success, -1 on cache miss, positive errno on failure.

cache_enter(*dvp*, *vp*, *cnp*)

Add an entry to the cache. **cache_enter**() is called with *dvp* pointing to the vnode of the directory to enter and *cnp* pointing to the partially initialised component structure. If *vp* is NULL, a negative cache entry is created, specifying that the entry does not exist in the file system. *cnp->cn_nameptr* points to the name of the entry being entered, *cnp->cn_namelen* tells the length of the name, and *cnp->cn_hash* contains a hash of the name.

cache_purge(*vp*)

Flush the cache of a particular vnode *vp*. **cache_purge**() is called when a vnode is renamed to hide entries that would now be invalid.

cache_purgevfs(*mp*)

Flush cache of a whole file system *mp*. **cache_purgevfs**() is called when file system is unmounted to remove entries that would now be invalid.

namecache_print(*vp*, *func*)

Print all entries of the name cache. *func* is the printf(9) format. **namecache_print**() is only defined if the kernel option DDB is compiled into the kernel.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the name lookup cache can be found. All pathnames are relative to /usr/src.

The name lookup cache is implemented within the file sys/kern/vfs_cache.c.

SEE ALSO

intro(9), namei(9), vfs(9), vnode(9)

HISTORY

The name lookup cache first appeared in 4.2BSD.

AUTHORS

The original name lookup cache was written by Robert Elz.

NAME

namei, **lookup**, **relookup**, **NDINIT** — pathname lookup

SYNOPSIS

```
#include <sys/namei.h>
#include <sys/uio.h>
#include <sys/vnode.h>

int
namei(struct nameidata *ndp);

int
lookup(struct nameidata *ndp);

int
relookup(struct vnode *dvp, struct vnode **vpp, struct componentname *cnp);

void
NDINIT(struct nameidata *ndp, u_long op, u_long flags, enum uio_seg segflg,
    const char *namep);
```

DESCRIPTION

The **namei** interface is used to convert pathnames to file system vnodes. The name of the interface is actually a contraction of the words *name* and *inode* for name-to-inode conversion, in the days before the vfs(9) interface was implemented.

The arguments passed to the functions are encapsulated in the *nameidata* structure. It has the following structure:

```
struct nameidata {
        /*
         * Arguments to namei/lookup.
         */
        const char *ni_dirp;            /* pathname pointer */
        enum    uio_seg ni_segflg;      /* location of pathname */
        /*
         * Arguments to lookup.
         */
        struct  vnode *ni_startdir;     /* starting directory */
        struct  vnode *ni_rootdir;      /* logical root directory */
        /*
         * Results: returned from/manipulated by lookup
         */
        struct  vnode *ni_vp;           /* vnode of result */
        struct  vnode *ni_dvp;          /* vnode of intermediate dir */
        /*
         * Shared between namei and lookup/commit routines.
         */
        size_t  ni_pathlen;             /* remaining chars in path */
        const char *ni_next;            /* next location in pathname */
        u_long  ni_loopcnt;             /* count of symlinks encountered */
        /*
         * Lookup parameters
         */
        struct componentname {
```

```
                /*
                 * Arguments to lookup.
                 */
                u_long  cn_nameiop;     /* namei operation */
                u_long  cn_flags;       /* flags to namei */
                kauth_cred_t cn_cred;   /* credentials */
                /*
                 * Shared between lookup and commit routines.
                 */
                char    *cn_pnbuf;      /* pathname buffer */
                const char *cn_nameptr; /* pointer to looked up name */
                long    cn_namelen;     /* length of looked up component */
                u_long  cn_hash;        /* hash value of looked up name */
                long    cn_consume;     /* chars to consume in lookup() */
        } ni_cnd;
};
```

The **namei** interface accesses vnode operations by passing arguments in the partially initialised *componentname* structure *ni_cnd*. This structure describes the subset of information from the nameidata structure that is passed through to the vnode operations. See vnodeops(9) for more information. The details of the componentname structure are not absolutely necessary since the members are initialised by the helper macro **NDINIT**(). It is useful to know the operations and flags as specified in vnodeops(9).

The **namei** interface overloads *ni_cnd.cn_flags* with some additional flags. These flags should be specific to the **namei** interface and ignored by vnode operations. However, due to the historic close relationship between the **namei** interface and the vnode operations, these flags are sometimes used (and set) by vnode operations, particularly **VOP_LOOKUP**(). The additional flags are:

NOCROSSMOUNT	do not cross mount points
RDONLY	lookup with read-only semantics
HASBUF	caller has allocated pathname buffer *ni_cnd.cn_pnbuf*
SAVENAME	save pathname buffer
SAVESTART	save starting directory
ISDOTDOT	current pathname component is ..
MAKEENTRY	add entry to the name cache
ISLASTCN	this is last component of pathname
ISSYMLINK	symlink needs interpretation
ISWHITEOUT	found whiteout
DOWHITEOUT	do whiteouts
REQUIREDIR	must be a directory
CREATEDIR	trailing slashes are ok
PARAMASK	mask of parameter descriptors

If the caller of **namei**() sets the SAVENAME flag, then it must free the buffer. If **VOP_LOOKUP**() sets the flag, then the buffer must be freed by either the commit routine or the **VOP_ABORT**() routine. The SAVESTART flag is set only by the callers of **namei**(). It implies SAVENAME plus the addition of saving the parent directory that contains the name in *ni_startdir*. It allows repeated calls to **lookup**() for the name being sought. The caller is responsible for releasing the buffer and for invoking **vrele**() on *ni_startdir*.

All access to the **namei** interface must be in process context. Pathname lookups cannot be done in interrupt context.

FUNCTIONS

namei(*ndp*)

Convert a pathname into a pointer to a vnode. The pathname is specified by *ndp->ni_dirp* and is of length *ndp->ni_pathlen*. The *ndp->segflg* flags defines whether the name in *ndp->ni_dirp* is an address in kernel space (UIO_SYSSPACE) or an address in user space (UIO_USERSPACE).

The vnode for the pathname is returned in *ndp->ni_vp*. The parent directory is returned locked in *ndp->ni_dvp* iff LOCKPARENT is specified.

If *ndp->ni_cnd.cn_flags* has the FOLLOW flag set then symbolic links are followed when they occur at the end of the name translation process. Symbolic links are always followed for all other pathname components other than the last.

lookup(*ndp*)

Search for a pathname. This is a very central and rather complicated routine.

The pathname is specified by *ndp->ni_dirp* and is of length *ndp->ni_pathlen*. The starting directory is taken from *ndp->ni_startdir*. The pathname is descended until done, or a symbolic link is encountered.

The semantics of **lookup**() are altered by the operation specified by *ndp->ni_cnd.cn_nameiop*. When CREATE, RENAME, or DELETE is specified, information usable in creating, renaming, or deleting a directory entry may be calculated.

If the target of the pathname exists and LOCKLEAF is set, the target is returned locked in *ndp->ni_vp*, otherwise it is returned unlocked.

relookup(*dvp*, *vpp*, *cnp*)

Reacquire a path name component is a directory. This is a quicker way to lookup a pathname component when the parent directory is known. The locked parent directory vnode is specified by *dvp* and the pathname component by *cnp*. The vnode of the pathname is returned in the address specified by *vpp*.

NDINIT(*ndp*, *op*, *flags*, *segflg*, *namep*)

Initialise a nameidata structure pointed to by *ndp* for use by the **namei** interface. It saves having to deal with the componentname structure inside *ndp*. The operation and flags are specified by *op* and *flags* respectively. These are the values to which *ndp->ni_cnd.cn_nameiop* and *ndp->ni_cnd.cn_flags* are respectively set. The segment flags which defines whether the pathname is in kernel address space or user address space is specified by *segflag*. The argument *namep* is a pointer to the pathname that *ndp->ni_dirp* is set to.

This routine stores the credentials of the calling thread (*curlwp*) in *ndp*. In the rare case that another set of credentials is required for the namei operation, *ndp->ni_cnd.cn_cred* must be set manually.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the name lookup subsystem can be found. All pathnames are relative to `/usr/src`.

The name lookup subsystem is implemented within the file `sys/kern/vfs_lookup.c`.

SEE ALSO

`intro`(9), `namecache`(9), `vfs`(9), `vnode`(9), `vnodeops`(9)

BUGS

It is unfortunate that much of the **namei** interface makes assumptions on the underlying vnode operations. These assumptions are an artefact of the introduction of the vfs interface to split a file system interface which was historically designed as a tightly coupled module.

NAME

opencrypto, **crypto_get_driverid**, **crypto_register**, **crypto_kregister**, **crypto_unregister**, **crypto_done**, **crypto_kdone**, **crypto_newsession**, **crypto_freesession**, **crypto_dispatch**, **crypto_kdispatch**, **crypto_getreq**, **crypto_freereq** — API for cryptographic services in the kernel

SYNOPSIS

```
#include <opencrypto/cryptodev.h>

int32_t
crypto_get_driverid(u_int32_t);

int
crypto_register(u_int32_t, int, u_int16_t, u_int32_t,
    int (*)(void *, u_int32_t *, struct cryptoini *),
    int (*)(void *, u_int32_t *), int (*)(u_int64_t),
    int (*)(struct cryptop *), void *);

int
crypto_kregister(u_int32_t, int, u_int32_t,
    int (*)(void *, struct cryptkop *, int), void *);

int
crypto_unregister(u_int32_t, int);

void
crypto_done(struct cryptop *);

void
crypto_kdone(struct cryptkop *);

int
crypto_newsession(u_int64_t *, struct cryptoini *, int);

int
crypto_freesession(u_int64_t);

int
crypto_dispatch(struct cryptop *);

int
crypto_kdispatch(struct cryptkop *);

struct cryptop *
crypto_getreq(int);

void
crypto_freereq(struct cryptop *);


#define EALG_MAX_BLOCK_LEN      16

struct cryptoini {
        int                cri_alg;
        int                cri_klen;
        int                cri_rnd;
        void              *cri_key;
        u_int8_t           cri_iv[EALG_MAX_BLOCK_LEN];
```

```
        struct cryptoini  *cri_next;
};

struct cryptodesc {
        int                crd_skip;
        int                crd_len;
        int                crd_inject;
        int                crd_flags;
        struct cryptoini   CRD_INI;
        struct cryptodesc *crd_next;
};

struct cryptop {
        TAILQ_ENTRY(cryptop) crp_next;
        u_int64_t          crp_sid;
        int                crp_ilen;
        int                crp_olen;
        int                crp_etype;
        int                crp_flags;
        void              *crp_buf;
        void              *crp_opaque;
        struct cryptodesc *crp_desc;
        int                (*crp_callback)(struct cryptop *);
        void              *crp_mac;
};

struct crparam {
        void         *crp_p;
        u_int           crp_nbits;
};

#define CRK_MAXPARAM    8

struct cryptkop {
        TAILQ_ENTRY(cryptkop) krp_next;
        u_int              krp_op;         /* ie. CRK_MOD_EXP or other */
        u_int              krp_status;     /* return status */
        u_short            krp_iparams;    /* # of input parameters */
        u_short            krp_oparams;    /* # of output parameters */
        u_int32_t         krp_hid;
        struct crparam     krp_param[CRK_MAXPARAM];    /* kvm */
        int               (*krp_callback)(struct cryptkop *);
};
```

DESCRIPTION

opencrypto is a framework for drivers of cryptographic hardware to register with the kernel so "consumers" (other kernel subsystems, and eventually users through an appropriate device) are able to make use of it. Drivers register with the framework the algorithms they support, and provide entry points (functions) the framework may call to establish, use, and tear down sessions. Sessions are used to cache cryptographic information in a particular driver (or associated hardware), so initialization is not needed with every request. Consumers of cryptographic services pass a set of descriptors that instruct the framework (and the drivers registered with it) of the operations that should be applied on the data (more than one cryptographic

operation can be requested).

Keying operations are supported as well. Unlike the symmetric operators described above, these sessionless commands perform mathematical operations using input and output parameters.

Since the consumers may not be associated with a process, drivers may not use condition variables: `condvar`(9). The same holds for the framework. Thus, a callback mechanism is used to notify a consumer that a request has been completed (the callback is specified by the consumer on an per-request basis). The callback is invoked by the framework whether the request was successfully completed or not. An error indication is provided in the latter case. A specific error code, `EAGAIN`, is used to indicate that a session number has changed and that the request may be re-submitted immediately with the new session number. Errors are only returned to the invoking function if not enough information to call the callback is available (meaning, there was a fatal error in verifying the arguments). No callback mechanism is used for session initialization and teardown.

The **`crypto_newsession`**() routine is called by consumers of cryptographic services (such as the `ipsec`(4) stack) that wish to establish a new session with the framework. On success, the first argument will contain the Session Identifier (SID). The second argument contains all the necessary information for the driver to establish the session. The third argument indicates whether a hardware driver should be used (1) or not (0). The various fields in the *`cryptoini`* structure are:

`cri_alg` Contains an algorithm identifier. Currently supported algorithms are:

```
CRYPTO_DES_CBC
CRYPTO_3DES_CBC
CRYPTO_BLF_CBC
CRYPTO_CAST_CBC
CRYPTO_SKIPJACK_CBC
CRYPTO_MD5_HMAC
CRYPTO_SHA1_HMAC
CRYPTO_RIPEMD160_HMAC
CRYPTO_MD5_KPDK
CRYPTO_SHA1_KPDK
CRYPTO_AES_CBC
CRYPTO_ARC4
CRYPTO_MD5
CRYPTO_SHA1
```

`cri_klen` Specifies the length of the key in bits, for variable-size key algorithms.

`cri_rnd` Specifies the number of rounds to be used with the algorithm, for variable-round algorithms.

`cri_key` Contains the key to be used with the algorithm.

`cri_iv` Contains an explicit initialization vector (IV), if it does not prefix the data. This field is ignored during initialization. If no IV is explicitly passed (see below on details), a random IV is used by the device driver processing the request.

`cri_next` Contains a pointer to another *`cryptoini`* structure. Multiple such structures may be linked to establish multi-algorithm sessions (`ipsec`(4) is an example consumer of such a feature).

The *`cryptoini`* structure and its contents will not be modified by the framework (or the drivers used). Subsequent requests for processing that use the SID returned will avoid the cost of re-initializing the hardware (in essence, SID acts as an index in the session cache of the driver).

crypto_freesession() is called with the SID returned by **crypto_newsession**() to disestablish the session.

crypto_dispatch() is called to process a request. The various fields in the *cryptop* structure are:

crp_sid Contains the SID.

crp_ilen Indicates the total length in bytes of the buffer to be processed.

crp_olen On return, contains the length of the result, not including *crd_skip*. For symmetric crypto operations, this will be the same as the input length.

crp_alloctype
Indicates the type of buffer, as used in the kernel malloc(9) routine. This will be used if the framework needs to allocate a new buffer for the result (or for re-formatting the input).

crp_callback This routine is invoked upon completion of the request, whether successful or not. It is invoked through the **crypto_done**() routine. If the request was not successful, an error code is set in the *crp_etype* field. It is the responsibility of the callback routine to set the appropriate spl(9) level.

crp_etype Contains the error type, if any errors were encountered, or zero if the request was successfully processed. If the EAGAIN error code is returned, the SID has changed (and has been recorded in the *crp_sid* field). The consumer should record the new SID and use it in all subsequent requests. In this case, the request may be re-submitted immediately. This mechanism is used by the framework to perform session migration (move a session from one driver to another, because of availability, performance, or other considerations).

Note that this field only makes sense when examined by the callback routine specified in *crp_callback*. Errors are returned to the invoker of **crypto_process**() only when enough information is not present to call the callback routine (i.e., if the pointer passed is NULL or if no callback routine was specified).

crp_flags Is a bitmask of flags associated with this request. Currently defined flags are:

CRYPTO_F_IMBUF The buffer pointed to by *crp_buf* is an mbuf chain.

crp_buf Points to the input buffer. On return (when the callback is invoked), it contains the result of the request. The input buffer may be an mbuf chain or a contiguous buffer (of a type identified by *crp_alloctype*), depending on *crp_flags*.

crp_opaque This is passed through the crypto framework untouched and is intended for the invoking application's use.

crp_desc This is a linked list of descriptors. Each descriptor provides information about what type of cryptographic operation should be done on the input buffer. The various fields are:

crd_skip The offset in the input buffer where processing should start.

crd_len How many bytes, after *crd_skip*, should be processed.

crd_inject Offset from the beginning of the buffer to insert any results. For encryption algorithms, this is where the initialization vector (IV) will be inserted when encrypting or where it can be found when decrypting (subject to *crd_flags*). For MAC algorithms, this is where the result of the keyed hash will be inserted.

crd_flags The following flags are defined:

CRD_F_ENCRYPT For encryption algorithms, this bit is set when encryption is required (when not set, decryption is performed).

CRD_F_IV_PRESENT For encryption algorithms, this bit is set when the IV already precedes the data, so the *crd_inject* value will be ignored and no IV will be written in the buffer. Otherwise, the IV used to encrypt the packet will be written at the location pointed to by *crd_inject*. The IV length is assumed to be equal to the blocksize of the encryption algorithm. Some applications that do special "IV cooking", such as the half-IV mode in ipsec(4), can use this flag to indicate that the IV should not be written on the packet. This flag is typically used in conjunction with the CRD_F_IV_EXPLICIT flag.

CRD_F_IV_EXPLICIT For encryption algorithms, this bit is set when the IV is explicitly provided by the consumer in the *crd_iv* fields. Otherwise, for encryption operations the IV is provided for by the driver used to perform the operation, whereas for decryption operations it is pointed to by the *crd_inject* field. This flag is typically used when the IV is calculated "on the fly" by the consumer, and does not precede the data (some ipsec(4) configurations, and the encrypted swap are two such examples).

CRD_F_COMP For compression algorithms, this bit is set when compression is required (when not set, decompression is performed).

CRD_INI This *cryptoini* structure will not be modified by the framework or the device drivers. Since this information accompanies every cryptographic operation request, drivers may re-initialize state on-demand (typically an expensive operation). Furthermore, the cryptographic framework may reroute requests as a result of full queues or hardware failure, as described above.

crd_next Point to the next descriptor. Linked operations are useful in protocols such as ipsec(4), where multiple cryptographic transforms may be applied on the same block of data.

crypto_getreq() allocates a *cryptop* structure with a linked list of as many *cryptodesc* structures as were specified in the argument passed to it.

crypto_freereq() deallocates a structure *cryptop* and any *cryptodesc* structures linked to it. Note that it is the responsibility of the callback routine to do the necessary cleanups associated with the opaque field in the *cryptop* structure.

crypto_kdispatch() is called to perform a keying operation. The various fields in the *crytokop* structure are:

krp_op	Operation code, such as CRK_MOD_EXP.
krp_status	Return code. This errno-style variable indicates whether there were lower level reasons for operation failure.
krp_iparams	Number of input parameters to the specified operation. Note that each operation has a (typically hardwired) number of such parameters.
krp_oparams	Number of output parameters from the specified operation. Note that each operation has a (typically hardwired) number of such parameters.
krp_kvp	An array of kernel memory blocks containing the parameters.
krp_hid	Identifier specifying which low-level driver is being used.
krp_callback	Callback called on completion of a keying operation.

DRIVER-SIDE API

The **crypto_get_driverid**(), **crypto_register**(), **crypto_kregister**(), **crypto_unregister**(), and **crypto_done**() routines are used by drivers that provide support for cryptographic primitives to register and unregister with the kernel crypto services framework. Drivers must first use the **crypto_get_driverid**() function to acquire a driver identifier, specifying the *flags* as an argument (normally 0, but software-only drivers should specify CRYPTOCAP_F_SOFTWARE). For each algorithm the driver supports, it must then call **crypto_register**(). The first argument is the driver identifier. The second argument is an array of CRYPTO_ALGORITHM_MAX + 1 elements, indicating which algorithms are supported. The last three arguments are pointers to three driver-provided functions that the framework may call to establish new cryptographic context with the driver, free already established context, and ask for a request to be processed (encrypt, decrypt, etc.) **crypto_unregister**() is called by drivers that wish to withdraw support for an algorithm. The two arguments are the driver and algorithm identifiers, respectively. Typically, drivers for pcmcia(4) crypto cards that are being ejected will invoke this routine for all algorithms supported by the card. If called with CRYPTO_ALGORITHM_ALL, all algorithms registered for a driver will be unregistered in one go and the driver will be disabled (no new sessions will be allocated on that driver, and any existing sessions will be migrated to other drivers). The same will be done if all algorithms associated with a driver are unregistered one by one.

The calling convention for the three driver-supplied routines is:

```
int (*newsession) (void *, u_int32_t *, struct cryptoini *);
int (*freesession) (void *, u_int64_t);
int (*process) (void *, struct cryptop *, int);
```

On invocation, the first argument to **newsession**() contains the driver identifier obtained via **crypto_get_driverid**(). On successfully returning, it should contain a driver-specific session identifier. The second argument is identical to that of **crypto_newsession**().

The **freesession**() routine takes as argument the SID (which is the concatenation of the driver identifier and the driver-specific session identifier). It should clear any context associated with the session (clear hardware registers, memory, etc.).

The **process**() routine is invoked with a request to perform crypto processing. This routine must not block, but should queue the request and return immediately. Upon processing the request, the callback routine should be invoked. In case of error, the error indication must be placed in the *crp_etype* field of the *cryptop* structure. The *hint* argument can be set to CRYPTO_HINT_MORE the there will be more request right after this request. When the request is completed, or an error is detected, the **process**() routine should invoke **crypto_done**(). Session migration may be performed, as mentioned previously.

The **kprocess**() routine is invoked with a request to perform crypto key processing. This routine must not block, but should queue the request and return immediately. Upon processing the request, the callback routine should be invoked. In case of error, the error indication must be placed in the *krp_status* field of the *cryptkop* structure. When the request is completed, or an error is detected, the **kprocess**() routine should invoke **crypto_kdone**().

RETURN VALUES

crypto_register(), **crypto_kregister**(), **crypto_unregister**(), **crypto_newsession**(), and **crypto_freesession**() return 0 on success, or an error code on failure. **crypto_get_driverid**() returns a non-negative value on error, and −1 on failure. **crypto_getreq**() returns a pointer to a *cryptop* structure and NULL on failure. **crypto_dispatch**() returns EINVAL if its argument or the callback function was NULL, and 0 otherwise. The callback is provided with an error code in case of failure, in the *crp_etype* field.

FILES

sys/crypto/crypto.c most of the framework code

SEE ALSO

condvar(9), ipsec(4), pcmcia(4), malloc(9)

Angelos D. Keromytis, Jason L. Wright, and Theo de Raadt, *The Design of the OpenBSD Cryptographic Framework*, *Usenix*, 2003, June 2003.

HISTORY

The cryptographic framework first appeared in OpenBSD 2.7 and was written by Angelos D. Keromytis ⟨angelos@openbsd.org⟩.

Sam Leffler ported the crypto framework to FreeBSD and made performance improvements.

Jonathan Stone ⟨jonathan@NetBSD.org⟩ ported the cryptoframe from FreeBSD to NetBSD. **opencrypto** first appeared in NetBSD 2.0.

BUGS

The framework currently assumes that all the algorithms in a **crypto_newsession**() operation must be available by the same driver. If that's not the case, session initialization will fail.

The framework also needs a mechanism for determining which driver is best for a specific set of algorithms associated with a session. Some type of benchmarking is in order here.

Multiple instances of the same algorithm in the same session are not supported. Note that 3DES is considered one algorithm (and not three instances of DES). Thus, 3DES and DES could be mixed in the same request.

A queue for completed operations should be implemented and processed at some software spl(9) level, to avoid overall system latency issues, and potential kernel stack exhaustion while processing a callback.

When SMP time comes, we will support use of a second processor (or more) as a crypto device (this is actually AMP, but we need the same basic support).

NAME

optstr_get — Options string management

SYNOPSIS

```
#include <sys/optstr.h>

bool
optstr_get(const char *optstr, const char *key, char *buf, size_t bufsize);
```

DESCRIPTION

An options string is a list of key/value pairs represented in textual form. Each pair is expressed as ''key=value'' and is separated from other pairs by one or more spaces. For example:

```
key1=value1 key2=value2 key3=value3
```

Options strings are used to pass information between userland programs and the kernel in a binary-agnostic way. This makes them endianness and ABI independent.

FUNCTIONS

The following functions are provided to manage options strings:

optstr_get(*optstr*, *key*, *buf*, *bufsize*)
: Scans the *optstr* options string looking for the key *key* and stores its value in the buffer pointed to by *buf* copying a maximum of *bufsize* bytes. Returns 'true' if the key was found or 'false' otherwise, in which case *buf* is left unmodified.

CODE REFERENCES

The options string management functions are implemented within the files `sys/kern/subr_optstr.c` and `sys/sys/optstr.h`.

HISTORY

Options strings appeared in NetBSD 4.0.

NAME

panic — Bring down system on fatal error

SYNOPSIS

```
#include <sys/types.h>
#include <sys/systm.h>

void
panic(const char *fmt, ...);
```

DESCRIPTION

The **panic**() function terminates the NetBSD system. The message *fmt* is a printf(3) style format string which is printed to the console and saved in the variable *panicstr* for later retrieval via core dump inspection. A newline character is added at the end automatically, and is thus not needed in the format string.

If a kernel debugger is installed, control is passed to it after the message is printed. If the kernel debugger is ddb(4), control may be passed to it, depending on the value of *ddb.onpanic*. See options(4) for more details on setting *ddb.onpanic*. If control is not passed through to ddb(4), a ddb(4)-specific function is used to print the kernel stack trace, and then control returns to **panic**().

If control remains in **panic**(), an attempt is made to save an image of system memory on the configured dump device.

If during the process of handling the panic, **panic**() is called again (from the filesystem synchronization routines, for example), the system is rebooted immediately without synchronizing any filesystems.

RETURN VALUES

The **panic**() function does not return.

SEE ALSO

sysctl(3), ddb(4), ipkdb(4), options(4), savecore(8), swapctl(8), sysctl(8)

NAME

PCI, **pci_activate**, **pci_conf_read**, **pci_conf_write**, **pci_conf_print**, **pci_conf_capture**, **pci_conf_restore**, **pci_find_device**, **pci_get_capability**, **pci_mapreg_type**, **pci_mapreg_map**, **pci_mapreg_info**, **pci_intr_map**, **pci_intr_string**, **pci_intr_evcnt**, **pci_intr_establish**, **pci_intr_disestablish**, **pci_get_powerstate**, **pci_set_powerstate**, **pci_vpd_read**, **pci_vpd_write**, **pci_make_tag**, **pci_decompose_tag**, **pci_findvendor**, **pci_devinfo**, **PCI_VENDOR**, **PCI_PRODUCT**, **PCI_REVISION** — Peripheral Component Interconnect

SYNOPSIS

```
#include <machine/bus.h>
#include <dev/pci/pcivar.h>
#include <dev/pci/pcireg.h>
#include <dev/pci/pcidevs.h>

int
pci_activate(pci_chipset_tag_t pc, pcitag_t tag, int reg, void *sc,
    int (*wakeup)(pci_chipset_tag_t pc, pcitag_t tag,
                  void *sc, pcireg_t reg));

pcireg_t
pci_conf_read(pci_chipset_tag_t pc, pcitag_t tag, int reg);

void
pci_conf_write(pci_chipset_tag_t pc, pcitag_t tag, int reg, pcireg_t val);

void
pci_conf_print(pci_chipset_tag_t pc, pcitag_t tag,
    void (*func)(pci_chipset_tag_t, pcitag_t, const pcireg_t *));

void
pci_conf_capture(pci_chipset_tag_t pc, pcitag_t tag,
    struct pci_conf_state *);

void
pci_conf_restore(pci_chipset_tag_t pc, pcitag_t tag,
    struct pci_conf_state *);

int
pci_find_device(struct pci_attach_args *pa,
    int (*func)(struct pci_attach_args *));

int
pci_get_capability(pci_chipset_tag_t pc, pcitag_t tag, int capid,
    int *offsetp, pcireg_t *valuep);

pcireg_t
pci_mapreg_type(pci_chipset_tag_t pc, pcitag_t tag, int reg);

int
pci_mapreg_map(struct pci_attach_args *pa, int reg, pcireg_t type,
    int busflags, bus_space_tag_t *tagp, bus_space_handle_t *handlep,
    bus_addr_t *basep, bus_size_t *sizep);

int
pci_mapreg_info(pci_chipset_tag_t pc, pcitag_t tag, int reg, pcireg_t type,
    bus_addr_t *basep, bus_size_t *sizep, int *flagsp);
```

int
pci_find_rom(*struct pci_attach_args *pa*, *bus_space_tag_t bst*, *bus_space_handle_t bsh*, *int code*, *bus_space_handle_t *handlep*, *bus_space_size_t *sizep*);

int
pci_intr_map(*struct pci_attach_args *pa*, *pci_intr_handle_t *ih*);

*const char **
pci_intr_string(*pci_chipset_tag_t pc*, *pci_intr_handle_t ih*);

*const struct evcnt **
pci_intr_evcnt(*pci_chipset_tag_t pc*, *pci_intr_handle_t ih*);

*void **
pci_intr_establish(*pci_chipset_tag_t pc*, *pci_intr_handle_t ih*, *int level*, *int (*handler)(void *)*, *void *arg*);

void
pci_intr_disestablish(*pci_chipset_tag_t pc*, *void *ih*);

int
pci_set_powerstate(*pci_chipset_tag_t pc*, *pcitag_t tag*, *pcireg_t newstate*);

int
pci_get_powerstate(*pci_chipset_tag_t pc*, *pcitag_t tag*, *pcireg_t *state*);

int
pci_vpd_read(*pci_chipset_tag_t pc*, *pcitag_t tag*, *int offset*, *int count*, *pcireg_t *data*);

int
pci_vpd_write(*pci_chipset_tag_t pc*, *pcitag_t tag*, *int offset*, *int count*, *pcireg_t *data*);

pcitag_t
pci_make_tag(*pci_chipset_tag_t pc*, *int bus*, *int device*, *int function*);

void
pci_decompose_tag(*pci_chipset_tag_t pc*, *pcitag_t tag*, *int *busp*, *int *devicep*, *int *functionp*);

*char **
pci_findvendor(*pcireg_t id*);

void
pci_devinfo(*pcireg_t id*, *pcireg_t class*, *int show*, *char *cp*, *size_t len*);

int
PCI_VENDOR(*pcireg_t id*);

int
PCI_PRODUCT(*pcireg_t id*);

int
PCI_REVISION(*pcireg_t id*);

DESCRIPTION

The machine-independent **PCI** subsystem provides support for PCI devices.

The PCI bus was initially developed by Intel in the early 1990's to replace the ISA bus for interfacing with their Pentium processor. The PCI specification is widely regarded as well designed, and the PCI bus has found widespread acceptance in machines ranging from Apple's PowerPC-based systems to Sun's UltraSPARC-based machines.

The PCI bus is a multiplexed bus, allowing addresses and data on the same pins for a reduced number of pins. Data transfers can be 8-bit, 16-bit or 32-bit. A 64-bit extended PCI bus is also defined. Multi-byte transfers are little-endian. The PCI bus operates up to 33MHz and any device on the bus can be the bus master.

AGP is a version of PCI optimised for high-throughput data rates, particularly for accelerated frame buffers.

The PCI bus is a "plug and play" bus, in the sense that devices can be configured dynamically by software. The PCI interface chip on a PCI device bus presents a small window of registers into the PCI configuration space. These registers contain information about the device such as the vendor and a product ID. The configuration registers can also be written to by software to alter how the device interfaces to the PCI bus. An important register in the configuration space is the Base Address Register (BAR). The BAR is written to by software to map the device registers into a window of processor address space. Once this mapping is done, the device registers can be accessed relative to the base address.

DATA TYPES

Drivers for devices attached to the **PCI** will make use of the following data types:

pcireg_t
: Configuration space register.

pci_chipset_tag_t
: Chipset tag for the PCI bus.

pcitag_t
: Configuration tag describing the location and function of the PCI device. It contains the tuple ⟨bus, device, function⟩.

pci_intr_handle_t
: The opaque handle describing an established interrupt handler.

struct pci_attach_args
: Devices have their identity recorded in this structure. It contains the following members:

```
bus_space_tag_t pa_iot;              /* pci i/o space tag */
bus_space_tag_t pa_memt;      /* pci mem space tag */
bus_dma_tag_t pa_dmat;        /* DMA tag */
pci_chipset_tag_t pa_pc;
int pa_flags;                 /* flags */
pcitag_t pa_tag;
pcireg_t pa_id;
pcireg_t pa_class;
```

struct pci_conf_state
: Stores the PCI configuration state of a device. It contains the following member:

```
pcireg_t reg[16];                    /* pci conf register */
```

FUNCTIONS

pci_activate(*pc*, *tag*, *sc*, *fun*)

Attempt to bring the device to state D0. If the device is not in the D0 state call *fun* to restore its state. If *fun* is NULL then restoring from state D3 is going to fail.

pci_conf_read(*pc*, *tag*, *reg*)

Read from register *reg* in PCI configuration space. The argument *tag* is the PCI tag for the current device attached to PCI chipset *pc*.

pci_conf_write(*pc*, *tag*, *reg*, *val*)

Write to register *reg* in PCI configuration space. The argument *tag* is the PCI tag for the current device attached to PCI chipset *pc*.

pci_conf_print(*pc*, *tag*, *func*)

Print out most of the registers in the PCI configuration for the device. The argument *tag* is the PCI tag for the current device attached to PCI chipset *pc*. The argument *func* is a function called by **pci_conf_print**() to print the device-dependent registers. This function is only useful for driver development and is usually wrapped in pre-processor declarations.

pci_conf_capture(*pc*, *tag*, *pcs*)

Capture PCI configuration space into structure *pcs*. The argument *tag* is the PCI tag for the current device attached to the PCI chipset *pc*.

pci_conf_restore(*pc*, *tag*, *pcs*)

Restores PCI configuration space from structure *pcs*. The argument *tag* is the PCI tag for the current device attached to the PCI chipset *pc*.

pci_find_device(*pa*, *func*)

Find a device using a match function on all probed busses. The argument *func* is called by **pci_find_device**() to match a device. The argument *pa* is filled in if the device is matched. **pci_find_device**() returns 1 if the device is matched, and zero otherwise. This function is specifically for use by LKMs (see lkm(4)) and its use otherwise is strongly discouraged.

pci_get_capability(*pc*, *tag*, *capid*, *offsetp*, *valuep*)

Parse the device capability list in configuration space looking for capability *capid*. If *offsetp* is not NULL, the register offset in configuration space is returned in *offsetp*. If *valuep* is not NULL, the value of the capability is returned in *valuep*. The argument *tag* is the PCI tag for the current device attached to PCI chipset *pc*. This function returns 1 if the capability was found. If the capability was not found, it returns zero, and *offsetp* and *valuep* remain unchanged.

pci_mapreg_type(*pc*, *tag*, *reg*)

Interrogates the Base Address Register (BAR) in configuration space specified by *reg* and returns the default (or current) mapping type. Valid returns values are:

PCI_MAPREG_TYPE_IO

The mapping is to I/O address space.

PCI_MAPREG_TYPE_MEM

The mapping is to memory address space.

PCI_MAPREG_TYPE_MEM | PCI_MAPREG_MEM_TYPE_64BIT

The mapping is to 64-bit memory address space.

PCI_MAPREG_TYPE_ROM

The mapping is to ROM. Note that in the current implementation, PCI_MAPREG_TYPE_ROM has the same numeric value as PCI_MAPREG_TYPE_MEM.

The argument *tag* is the PCI tag for the current device attached to PCI chipset *pc*.

pci_mapreg_map(*pa*, *reg*, *type*, *busflags*, *tagp*, *handlep*, *basep*, *sizep*)
Maps the register windows for the device into kernel virtual address space. This function is generally only called during the driver attach step and takes a pointer to the *struct pci_attach_args* in *pa*. The physical address of the mapping is in the Base Address Register (BAR) in configuration space specified by *reg*. Valid values for the type of mapping *type* are:

PCI_MAPREG_TYPE_IO
The mapping should be to I/O address space.

PCI_MAPREG_TYPE_MEM
The mapping should be to memory address space.

PCI_MAPREG_TYPE_ROM
The mapping is to access ROM. This type of mapping is only permitted when the value for *reg* is PCI_MAPREG_ROM.

The argument *busflags* are bus-space flags passed to **bus_space_map**() to perform the mapping (see bus_space(9)). The bus-space tag and handle for the mapped register window are returned in *tagp* and *handlep* respectively. The bus-address and size of the mapping are returned in *basep* and *sizep* respectively. If any of *tagp*, *handlep*, *basep*, or *sizep* are NULL then **pci_mapreg_map**() does not define their return value. This function returns zero on success and non-zero on error.

pci_mapreg_info(*pc*, *tag*, *reg*, *type*, *basep*, *sizep*, *flagsp*)
Performs the same operations as **pci_mapreg_map**() but doesn't actually map the register window into kernel virtual address space. Returns the bus-address, size and bus flags in *basep*, *sizep* and *flagsp* respectively. These return values can be used by **bus_space_map**() to actually map the register window into kernel virtual address space. This function is useful for setting up the registers in configuration space and deferring the mapping to a later time, such as in a bus-independent attachment routine. *pci_mapreg_info* returns zero on success and non-zero on failure.

pci_find_rom(*pa*, *bst*, *bsh*, *code*, *handlep*, *sizep*)
Locates a suitable ROM image within a PCI expansion ROM previously mapped with **pci_mapreg_map**() and creates a subregion for it with **bus_space_subregion**(). The *bst* and *bsh* arguments are the bus tag and handle obtained with the prior call to **pci_mapreg_map**(). Valid values for the image type *code* are:

PCI_ROM_CODE_TYPE_X86
Find a ROM image containing i386 executable code for use by PC BIOS.

PCI_ROM_CODE_TYPE_OFW
Find a ROM image containing Forth code for use by Open Firmware.

PCI_ROM_CODE_TYPE_HPPA
Find a ROM image containing HP PA/RISC executable code.

The created subregion will cover the entire selected ROM image, including header data. The handle to this subregion is returned in *handlep*. The size of the image (and the corresponding subregion) is returned in *sizep*. This function can only be used with expansion ROMs located at the PCI_MAPREG_ROM base address register (BAR).

pci_intr_map(*pa*, *ih*)
See pci_intr(9).

pci_intr_string(*pc*, *ih*)
See pci_intr(9).

pci_intr_evcnt(*pc*, *ih*)
See pci_intr(9).

pci_intr_establish(*pc*, *ih*, *level*, *handler*, *arg*)
See pci_intr(9).

pci_intr_disestablish(*pc*, *ih*)
See pci_intr(9).

pci_set_powerstate(*pc*, *tag*, *newstate*)
Set power state of the device to newstate. Valid values for *newstate* are:

```
PCI_PMCSR_STATE_D0
PCI_PMCSR_STATE_D1
PCI_PMCSR_STATE_D2
PCI_PMCSR_STATE_D3
```

pci_get_powerstate(*pc*, *tag*, *state*)
Get current power state of the device.

pci_vpd_read(*pc*, *tag*, *offset*, *count*, *data*)
Read *count* 32-bit words of Vital Product Data for the device starting at offset *offset* into the buffer pointed to by *data*. Returns 0 on success or non-zero if the device has no Vital Product Data capability or if reading the Vital Product Data fails.

pci_vpd_write(*pc*, *tag*, *offset*, *count*, *data*)
Write *count* 32-bit words of Vital Product Data for the device starting at offset *offset* from the buffer pointed to by *data*. Returns 0 on success or non-zero if the device has no Vital Product Data capability of if writing the Vital Product Data fails.

pci_make_tag(*pc*, *bus*, *device*, *function*)
Create a new PCI tag for the PCI device specified by the tuple ⟨bus, device, function⟩. This function is not useful to the usual PCI device driver. It is generally used by drivers of multi-function devices when attaching other PCI device drivers to each function.

pci_decompose_tag(*pc*, *tag*, *busp*, *devicep*, *fnp*)
Decompose the PCI tag *tag* generated by **pci_make_tag**() into its ⟨bus, device, function⟩ tuple.

pci_findvendor(*id*)
Return the string of the vendor name for the device specified by *id*.

pci_devinfo(*id*, *class*, *show*, *cp*, *len*)
Returns the description string from the in-kernel PCI database for the device described by *id* and *class*. The description string is returned in *cp*; the size of that storage is given in *len*. The argument *show* specifies whether the PCI subsystem should report the string to the console.

PCI_VENDOR(*id*)
Return the PCI vendor id for device *id*.

PCI_PRODUCT(*id*)
Return the PCI product id for device *id*.

PCI_REVISION(*id*)
Return the PCI product revision for device *id*.

AUTOCONFIGURATION

During autoconfiguration, a **PCI** driver will receive a pointer to *struct pci_attach_args* describing the device attaches to the PCI bus. Drivers match the device using the *pa_id* member using **PCI_VENDOR**(). **PCI_PRODUCT**() and **PCI_REVISION**().

During the driver attach step, drivers can read the device configuration space using **pci_conf_read**(). The meaning attached to registers in the PCI configuration space are device-dependent, but will usually contain physical addresses of the device register windows. Device options can also be stored into the PCI configuration space using **pci_conf_write**(). For example, the driver can request support for bus-mastering DMA by writing the option to the PCI configuration space.

Device capabilities can be queried using **pci_get_capability**(), and returns device-specific information which can be found in the PCI configuration space to alter device operation.

After reading the physical addresses of the device register windows from configuration space, these windows must be mapped into kernel virtual address space using **pci_mapreg_map**(). Device registers can now be accessed using the standard bus-space API (see bus_space(9)).

Details of using PCI interrupts is described in pci_intr(9).

DMA SUPPORT

The PCI bus supports bus-mastering operations from any device on the bus. The DMA facilities are accessed through the standard bus_dma(9) interface. To support DMA transfers from the device to the host, it is necessary to enable bus-mastering in the PCI configuration space for the device.

During system shutdown, it is necessary to abort any DMA transfers in progress by registering a shutdown hook (see shutdownhook_establish(9)).

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent PCI subsystem can be found. All pathnames are relative to /usr/src.

The PCI subsystem itself is implemented within the files sys/dev/pci/pci.c, sys/dev/pci/pci_subr.c, sys/dev/pci/pci_map.c, sys/dev/pci/pci_quirks.c, and sys/dev/pci/pciconf.c. Machine-dependent portions are implemented within the file sys/arch/<arch>/pci/pci_machdep.c.

The database of known devices exists within the file sys/dev/pci/pcidevs_data.h and is generated automatically from the file sys/dev/pci/pcidevs. New vendor and product identifiers should be added to this file. The database can be regenerated using the Makefile sys/dev/pci/Makefile.pcidevs.

SEE ALSO

pci(4), autoconf(9), bus_dma(9), bus_space(9), driver(9), pci_configure_bus(9), pci_intr(9), shutdownhook_establish(9)

HISTORY

The machine-independent PCI subsystem appeared in NetBSD 1.2.

NAME

pci_configure_bus, **pci_conf_hook**, **pci_conf_interrupt** — perform PCI bus configuration

SYNOPSIS

```
#include <dev/pci/pciconf.h>

int
pci_configure_bus(pci_chipset_tag_t pc, struct extent *ioext,
    struct extent *memext, struct extent *pmemext, int firstbus,
    int cacheline_size);
```

DESCRIPTION

The **pci_configure_bus**() function configures a PCI bus for use. This involves:

- Defining bus numbers for all busses on the system,
- Setting the Base Address Registers for all devices,
- Setting up the interrupt line register for all devices,
- Configuring bus latency timers for all devices, and
- Configuring cacheline sizes for all devices.

In traditional PCs and Alpha systems, the BIOS or firmware takes care of this task, but that is not the case for all systems. **pci_configure_bus**() should be called prior to the autoconfiguration of the bus.

The *pc* argument is a machine-dependent tag used to specify the PCI chipset to the system. This should be the same value used with **pci_make_tag**(). The extent arguments define memory extents from which the address space for the cards will be taken. These addresses should be in the PCI address space. The *ioext* extent is for PCI I/O accesses. The *memext* extent is for PCI memory accesses that might have side effects. I.e., that can not be cached. The *pmemext* extent is for PCI memory accesses that can be cached. The *pmemext* extent will be used for any ROMs and any memory regions that are marked as "prefetchable" in their BAR. If an implementation does not distinguish between prefetchable and non-prefetchable memory, it may pass NULL for *pmemext*. In this case, prefetchable memory allocations will be made from the non-prefetchable region. The *firstbus* argument indicates the number of the first bus to be configured. The *cacheline_size* argument is used to configure the PCI Cache Line Size Register; it should be the size, in bytes, of the largest D-cache line on the system.

An implementation may choose to not have full configuration performed by **pci_configure_bus**() on certain PCI devices, such as PCI host bridges or PCI bus analyzers which are instantiated as devices on the bus. In order for this to take place, the header ⟨machine/pci_machdep.h⟩ must define the __HAVE_PCI_CONF_HOOK symbol (without a value), and a machine-dependent function **pci_conf_hook**() (declared in the same header) must be defined. The prototype for this function is

int pci_conf_hook(*pci_chipset_tag_t pc*, *int bus*, *int device*, *int function*, *pcireg_t id*)

In this function, *bus*, *device*, and *function* uniquely identify the item being configured; in addition to this, the value of the device's PCI identification register is passed in *id*. For each device **pci_conf_hook**() can then decide upon the amount of configuration to be performed by returning a bitwise inclusive-or of the following flags:

PCI_CONF_MAP_IO	Configure Base Address Registers that map I/O space
PCI_CONF_MAP_MEM	Configure Base Address Registers that map memory space

`PCI_CONF_MAP_ROM`	Configure Expansion ROM Base Address register
`PCI_CONF_ENABLE_IO`	Enable I/O space accesses
`PCI_CONF_ENABLE_MEM`	Enable memory space accesses
`PCI_CONF_ENABLE_BM`	Enable bus mastering

In addition, `PCI_CONF_ALL` specifies all of the above.

One of the functions of **pci_configure_bus**() is to configure interrupt "line" information. This must be done on a machine-dependent basis, so a machine-dependent function **pci_conf_interrupt**() must be defined. The prototype for this function is

void pci_conf_interrupt(*pci_chipset_tag_t pc*, *int bus*, *int device*, *int pin*, *int swiz*, *int *iline*)

In this function, *bus*, *device*, and *pin*, uniquely identify the item being configured. The *swiz* argument is a "swizzle", a sum of the device numbers of the primary interface of the bridges between the host bridge and the current device. The function is responsible for setting the value of *iline*. See chapter 9 of the "PCI-to-PCI Bridge Architecture Specification" for more information on swizzling (also known as interrupt routing).

RETURN VALUES

If successful **pci_configure_bus**() returns 0. A non-zero return value means that the bus was not completely configured for some reason. A description of the failure will be displayed on the console.

ENVIRONMENT

The **pci_configure_bus**() function is only included in the kernel if the kernel is compiled with the `PCI_NETBSD_CONFIGURE` option enabled.

EXAMPLES

The **pci_conf_hook**() function in evbppc's walnut implementation looks like:

```
int
pci_conf_hook(pci_chipset_tag_t pc, int bus, int dev, int func,
    pcireg_t id)
{

        if ((PCI_VENDOR(id) == PCI_VENDOR_IBM &&
             PCI_PRODUCT(id) == PCI_PRODUCT_IBM_405GP) ||
            (PCI_VENDOR(id) == PCI_VENDOR_INTEL &&
             PCI_PRODUCT(id) == PCI_PRODUCT_INTEL_80960_RP)) {
                /* Don't configure the bridge and PCI probe. */
                return 0;
        }
        return (PCI_CONF_ALL & ~PCI_CONF_MAP_ROM);
}
```

The **pci_conf_interrupt**() function in the sandpoint implementation looks like:

```
void
pci_conf_interrupt(pci_chipset_tag_t pc, int bus, int dev, int pin,
    int swiz, int *iline)
{
        if (bus == 0) {
                *iline = dev;
```

```
        } else {
                *iline = 13 + ((swiz + dev + 3) & 3);
        }
}
```

The BeBox has nearly 1GB of PCI I/O memory starting at processor address 0x81000000 (PCI I/O address 0x01000000), and nearly 1GB of PCI memory starting at 0xC0000000 (PCI memory address 0x00000000). The **pci_configure_bus**() function might be called as follows:

```
struct extent *ioext, *memext;
...
ioext  = extent_create("pciio",  0x01000000, 0x0fffffff, M_DEVBUF,
    NULL, 0, EX_NOWAIT);
memext = extent_create("pcimem", 0x00000000, 0x0fffffff, M_DEVBUF,
    NULL, 0, EX_NOWAIT);
...
pci_configure_bus(0, ioext, memext, NULL);
...
extent_destroy(ioext);
extent_destroy(memext);
...
```

Note that this must be called before the PCI bus is attached during autoconfiguration.

SEE ALSO

pci(4), extent(9)

HISTORY

pci_configure_bus() was added in NetBSD 1.6.

NAME

pci_intr, **pci_intr_map**, **pci_intr_string**, **pci_intr_establish**, **pci_intr_disestablish** — PCI bus interrupt manipulation functions

SYNOPSIS

```
#include <dev/pci/pcivar.h>

int
pci_intr_map(struct pci_attach_args *pa, pci_intr_handle_t *ih);

const char *
pci_intr_string(pci_chipset_t *pc, pci_intr_handle_t ih);

void *
pci_intr_establish(pci_chipset_t *pc, pci_intr_handle_t ih, int ipl,
    int (*intrhand)(void *), void *intrarg);

void
pci_intr_disestablish(pci_chipset_t *pc, void *ih);
```

DESCRIPTION

The **pci_intr** functions exist to allow device drivers machine-independent access to PCI bus interrupts. The functions described in this page are typically declared in a port's ⟨machine/pci_machdep.h⟩ header file; however, drivers should generally include ⟨dev/pci/pcivar.h⟩ to get other PCI-specific declarations as well.

Each driver has an **attach**() function which has a bus-specific *attach_args* structure. Each driver for a PCI device is passed a pointer to an object of type *struct pci_attach_args* which contains, among other things, information about the location of the device in the PCI bus topology sufficient to allow interrupts from the device to be handled.

If a driver wishes to establish an interrupt handler for the device, it should pass the *struct pci_attach_args* * to the **pci_intr_map**() function, which returns zero on success, and nonzero on failure. The function sets the *pci_intr_handle_t* pointed at by its second argument to a machine-dependent value which identifies a particular interrupt source.

If the driver wishes to refer to the interrupt source in an attach or error message, it should use the value returned by **pci_intr_string**().

Subsequently, when the driver is prepared to receive interrupts, it should call **pci_intr_establish**() to actually establish the handler; when the device interrupts, *intrhand* will be called with a single argument *intrarg*, and will run at the interrupt priority level *ipl*.

The return value of **pci_intr_establish**() may be saved and passed to **pci_intr_disestablish**() to disable the interrupt handler when the driver is no longer interested in interrupts from the device.

PORTING

A port's implementation of **pci_intr_map**() may use the following members of *struct pci_attach_args* to determine how the device's interrupts are routed.

```
pci_chipset_tag_t pa_pc;
pcitag_t pa_tag;
pcitag_t pa_intrtag; /* intr. appears to come from here */
pci_intr_pin_t pa_intrpin; /* intr. appears on this pin */
pci_intr_line_t pa_intrline; /* intr. routing information */
pci_intr_pin_t pa_rawintrpin; /* unswizzled pin */
```

PCI-PCI bridges swizzle (permute) interrupt wiring. Depending on implementation details, it may be more convenient to use either original or the swizzled interrupt parameters. The original device tag and interrupt pin can be found in *pa_tag* and *pa_rawintrpin* respectively, while the swizzled tag and pin can be found in *pa_intrtag* and *pa_intrpin*.

When a device is attached to a primary bus, both pairs of fields contain the same values. When a device is found behind one or more pci-pci bridges, *pa_intrpin* contains the "swizzled" interrupt pin number, while *pa_rawintrpin* contains the original interrupt pin; *pa_tag* contains the PCI tag of the device itself, and *pa_intrtag* contains the PCI tag of the uppermost bridge device.

NAME

pckbport, **pckbport_attach**, **pckbport_attach_slot**, **pckbport_cnattach**, **pckbportintr**, **pckbport_set_inputhandler**, **pckbport_flush**, **pckbport_poll_cmd**, **pckbport_enqueue_cmd**, **pckbport_poll_data**, **pckbport_set_poll**, **pckbport_xt_translation**, **pckbport_slot_enable** — PC keyboard port interface

SYNOPSIS

```
#include <dev/pckbport/pckbportvar.h>

pckbport_tag_t
pckbport_attach(void *, struct pckbport_accessops const *);

struct device *
pckbport_attach_slot(struct device *, pckbport_tag_t, pckbport_slot_t);

int
pckbport_cnattach(void *, struct pckbport_accessops const *,
    pckbport_slot_t);

void
pckbportintr(pckbport_tag_t, pckbport_slot_t, int);

void
pckbport_set_inputhandler(pckbport_tag_t, pckbport_slot_t,
    pckbport_inputfcn, void *, char *);

void
pckbport_flush(pckbport_tag_t, pckbport_slot_t);

int
pckbport_poll_cmd(pckbport_tag_t, pckbport_slot_t, u_char *, int, int,
    u_char *, int);

int
pckbport_enqueue_cmd(pckbport_tag_t, pckbport_slot_t, u_char *, int, int,
    int, u_char *);

int
pckbport_poll_data(pckbport_tag_t, pckbport_slot_t);

void
pckbport_set_poll(pckbport_tag_t, pckbport_slot_t, int);

int
pckbport_xt_translation(pckbport_tag_t, pckbport_slot_t, int);

void
pckbport_slot_enable(pckbport_tag_t, pckbport_slot_t, int);
```

DESCRIPTION

The machine-independent **pckbport** subsystem provides an interface layer corresponding to the serial keyboard and mouse interface used on the IBM PS/2 and many other machines. It interfaces a controller driver such as pckbc(4) to device drivers such as pckbd(4) and pms(4).

A single controller can have up to two ports (known as "slots"), and these are identified by values of type *pckbport_slot_t*. The values PCKBPORT_KBD_SLOT and PCKBPORT_AUX_SLOT should be used for keyboard and mouse ports respectively. Each controller is identified by an opaque value of type *pckbport_tag_t*.

Controller interface

A PC keyboard controller registers itself by calling **pckbport_attach**(*cookie*, *ops*), with *ops* being a pointer to a *struct pckbport_accessops* containing pointers to functions for driving the controller, which will all be called with *cookie* as their first argument. **pckbport_attach**() returns the *pckbport_tag_t* assigned to the controller. The controller is then expected to call **pckbport_attach_slot**() for each slot with which it is equipped, passing the *struct device* * corresponding to the controller. This function returns a pointer to the child device attached to the slot, or NULL if no such device was attached.

The elements of *struct pckbport_accessops* each take as their first two arguments the *cookie* passed to **pckbport_attach**() and the slot in question. The elements are:

int **(*t_xt_translation)**(*void *cookie*, *pckbport_slot_t slot*, *int on*)
: If *on* is non-zero, enable, otherwise disable, AT-to-XT keycode translation on the slot specified. Returns 1 on success, 0 on failure (or if the controller does not support such translation).

int **(*t_send_devcmd)**(*void *cookie*, *pckbport_slot_t slot*, *u_char byte*)
: Send a single *byte* to the device without waiting for completion. Returns 1 on success, 0 on failure.

int **(*t_poll_data1)**(*void *cookie*, *pckbport_slot_t slot*)
: Wait for and return one byte of data from the device, without using interrupts. This function will only be called after **(*t_set_poll)**() has been used to put the slot in polling mode. If no data are forthcoming from the device after about 100ms, return −1.

void **(*t_slot_enable)**(*void *cookie*, *pckbport_slot_t slot*, *int on*)
: If *on* is non-zero, enable, otherwise disable, the slot. If a slot is disabled, it can be powered down, and is not expected to generate any interrupts. When first attached, ports should be disabled.

void **(*t_intr_establish)**(*void *cookie*, *pckbport_slot_t slot*)
: Set up an interrupt handler for the slot. Called when a device gets attached to it.

void **(*t_set_poll)**(*void *cookie*, *pckbport_slot_t slot*, *int on*)
: If *on* is non-zero, enable, otherwise disable, polling mode on the slot. In polling mode, data received from the device are provided to **(*t_poll_data1)**() and not passed to **pckbportintr**(), whether or not interrupts are enabled. In non-polling mode, data from the device are expected to cause interrupts. The controller interrupt handler should call **pckbportintr**(*tag*, *slot*, *byte*) once for each *byte* received from the device. When first attached, a port should be in non-polling mode.

Device interface

Devices that attach to **pckbport** controllers do so using the normal autoconf(9) mechanism. Their **(*ca_match)**() and **(*ca_attach)**() functions get passed a *struct pckbport_attach_args* which contains the controller and slot number where the device was found. Device drivers can use the following functions to communicate with the controller. Each takes *tag* and *slot* arguments to specify the slot to be acted on.

pckbport_set_inputhandler(*tag*, *slot*, *fn*, *arg*, *name*)
: Arrange for *fn* to be called with argument *arg* whenever an unsolicited byte is received from the slot. The function will be called at **spltty**().

pckbport_flush(*tag*, *slot*)
: Ensure that there is no pending input from the slot.

pckbport_poll_cmd(*tag*, *slot*, *cmd*, *len*, *responselen*, *respbuf*, *slow*)
Issue a complete device command, *cmd*, *len* bytes long, expecting a response *responselen* bytes long, which will be placed in *respbuf*. If *slow* is true, the command is expected to take over a second to execute. **pckbport_poll_cmd**() handles getting an acknowledgement from the device and retrying the command if necessary. Returns 0 on success, and an error value on failure. This function should only be called during autoconfiguration or when the slot has been placed into polling mode by **pckbport_set_poll**().

pckbport_enqueue_cmd(*tag*, *slot*, *cmd*, *len*, *responselen*, *sync*, *respbuf*)
Issue a complete device command, *cmd*, *len* bytes long, expecting a response *responselen* bytes long, which will be places in *respbuf*. If *sync* is true, **pckbport_enqueue_cmd**() waits for the command to complete before returning, otherwise it returns immediately. It is not safe to set *sync* when calling from an interrupt context. The **pckbport** layer handles getting an acknowledgement from the device and retrying the command if necessary. Returns 0 on success, and an error value on failure.

pckbport_poll_data(*tag*, *slot*)
Low-level command to poll for a single byte of data from the device, but ignoring bytes that are part of the response to a command issued through **pckbport_enqueue_command**().

pckbport_set_poll(*tag*, *slot*, *on*)
If *on* is true, enable polling on the slot, otherwise disable it. In polling mode, **pckbport_poll_cmd**() can be used to issue commands and **pckbport_poll_data**() to read unsolicited data, without enabling interrupts. In non-polling mode, commands should be issued using **pckbport_enqueue_cmd**(), unsolicited data are handled by the input function, and disabling interrupts will suspend **pckbport** operation.

pckbport_xt_translation(*tag*, *slot*, *on*)
Passthrough of **(*t_xt_translation)**() (see above).

pckbport_slot(*enable*, *tag*, *slot*, *on*)
Passthrough of **(*t_slot_enable)**() (see above).

Console interface

On systems that can attach consoles through **pckbport**, the controller's console attachment function (called very early in autoconfiguration) calls **pckbport_cnattach**(*cookie*, *ops*, *slot*). The first two arguments are the same as for **pckbport_attach**(), while the third indicates which slot the console keyboard is attached to. **pckbport_cnattach**() either calls **pckbd_cnattach**(), if it is available, or **pckbport_machdep_cnattach**(). The latter allows machine-dependent keyboard drivers to attach themselves, but it is only called if a device with the pckbport_machdep_cnattach attribute is configured into the system. **pckbport_cnattach**() returns 0 on success and an error value on failure. **pckbport_machdep_cnattach**() is expected to do the same.

CODE REFERENCES

The **pckbport** code, and the pckbd(4) and pms(4) device drivers are in sys/dev/pckbport.

SEE ALSO

pckbc(4), pckbd(4), pms(4), autoconf(9), spl(9)

HISTORY

The **pckbport** system appeared in NetBSD 2.0. Before that, pckbd(4) and pms(4) attached directly to pckbc(4) without any sensible way of using a different controller.

NAME

PCMCIA **pcmcia_function_init**, **pcmcia_function_enable**, **pcmcia_function_disable**, **pcmcia_io_alloc**, **pcmcia_io_free**, **pcmcia_io_map**, **pcmcia_io_unmap**, **pcmcia_mem_alloc**, **pcmcia_mem_free**, **pcmcia_mem_map**, **pcmcia_mem_unmap**, **pcmcia_intr_establish**, **pcmcia_intr_disestablish**, **pcmcia_cis_read_1**, **pcmcia_cis_read_2**, **pcmcia_cis_read_3**, **pcmcia_cis_read_4**, **pcmcia_cis_read_n**, **pcmcia_scan_cis** — support for PCMCIA PC-Card devices

SYNOPSIS

```
#include <machine/bus.h>
#include <dev/pcmcia/pcmciareg.h>
#include <dev/pcmcia/pcmciavar.h>
#include <dev/pcmcia/pcmciadevs.h>

void
pcmcia_function_init(struct pcmcia_function *pf,
    struct pcmcia_config_entry *cfe);

int
pcmcia_function_enable(struct pcmcia_function *pf);

void
pcmcia_function_disable(struct pcmcia_function *pf);

int
pcmcia_io_alloc(struct pcmcia_function *pf, bus_addr_t start,
    bus_size_t size, bus_size_t align, struct pcmcia_io_handle *pciop);

void
pcmcia_io_free(struct pcmcia_function *pf, struct pcmcia_io_handle *pcihp);

int
pcmcia_io_map(struct pcmcia_function *pf, int width,
    struct pcmcia_io_handle *pcihp, int *windowp);

void
pcmcia_io_unmap(struct pcmcia_function *pf, int window);

int
pcmcia_mem_alloc(struct pcmcia_function *pf, bus_size_t size,
    struct pcmcia_mem_handle *pcmhp);

void
pcmcia_mem_free(struct pcmcia_function *pf,
    struct pcmcia_mem_handle *pcmhp);

int
pcmcia_mem_map(struct pcmcia_function *pf, int width, bus_addr_t card_addr,
    bus_size_t size, struct pcmcia_mem_handle *pcmhp, bus_size_t *offsetp,
    int *windowp);

void
pcmcia_mem_unmap(struct pcmcia_function *pf, int window);

void *
pcmcia_intr_establish(struct pcmcia_function *pf, int level,
    int (*handler)(void *), void *arg);
```

void
pcmcia_intr_disestablish(*struct pcmcia_function *pf*, *void *ih*);

uint8_t
pcmcia_cis_read_1(*struct pcmcia_tuple *tuple*, *int index*);

uint16_t
pcmcia_cis_read_2(*struct pcmcia_tuple *tuple*, *int index*);

uint32_t
pcmcia_cis_read_3(*struct pcmcia_tuple *tuple*, *int index*);

uint32_t
pcmcia_cis_read_4(*struct pcmcia_tuple *tuple*, *int index*);

uint32_t
pcmcia_cis_read_n(*struct pcmcia_tuple *tuple*, *int number*, *int index*);

int
pcmcia_scan_cis(*struct device *dev*,
    *int (*func)(struct pcmcia_tuple *, void *)*, *void *arg*);

DESCRIPTION

The machine-independent **PCMCIA** subsystem provides support for PC-Card devices defined by the Personal Computer Memory Card International Assocation (PCMCIA). The **PCMCIA** bus supports insertion and removal of cards while a system is powered-on (ie, dynamic reconfiguration). The socket must be powered-off when a card is not present. To the user, this appears as though the socket is "hot" during insertion and removal events.

A PCMCIA controller interfaces the PCMCIA bus with the ISA or PCI busses on the host system. The controller is responsible for detecting and enabling devices and for allocating and mapping resources such as memory and interrupts to devices on the PCMCIA bus.

Each device has a table called the Card Information Structure (CIS) which contains configuration information. The tuples in the CIS are used by the controller to uniquely identify the device. Additional information may be present in the CIS, such as the ethernet MAC address, that can be accessed and used within a device driver.

Devices on the PCMCIA bus are uniquely identified by a 32-bit manufacturer ID and a 32-bit product ID. Additionally, devices can perform multiple functions (such as ethernet and modem) and these functions are identified by a function ID.

PCMCIA devices do not support DMA, however memory on the device can be mapped into the address space of the host.

DATA TYPES

Drivers attached to the **PCMCIA** bus will make use of the following data types:

struct pcmcia_card
Devices (cards) have their identity recorded in this structure. It contains the following members:

```
char           *cis1_info[4];
int32_t         manufacturer;
int32_t         product;
uint16_t       error;
SIMPLEQ_HEAD(, pcmcia_function)      pf_head;
```

struct pcmcia_function
: Identifies the function of the devices. A device can have multiple functions. Consider it an opaque type for identifying a particular function of a device.

struct pcmcia_config_entry
: Contains information about the resources requested by the device. It contains the following members:

```
int             number;
uint32_t       flags;
int            iftype;
int            num_iospace;
u_long                iomask;
struct {
        u_long  length;
        u_long  start;
} iospace[4];
uint16_t       irqmask;
int             num_memspace;
struct {
        u_long  length;
        u_long  cardaddr;
        u_long  hostaddr;
}              memspace[2];
int             maxtwins;
SIMPLEQ_ENTRY(pcmcia_config_entry) cfe_list;
```

struct pcmcia_tuple
: A handle for identifying an entry in the CIS.

struct pcmcia_io_handle
: A handle for mapping and allocating I/O address spaces. It contains the tag and handle for accessing the bus-space.

struct pcmcia_mem_handle
: A handle for mapping and allocating memory address spaces. It contains the tag and handle for accessing the bus-space.

struct pcmcia_attach_args
: A structure used to inform the driver of the device properties. It contains the following members:

```
int32_t               manufacturer;
int32_t               product;
struct pcmcia_card    *card;
struct pcmcia_function *pf;
```

FUNCTIONS

pcmcia_function_init(*pf*, *cfe*)
: Initialise the machine-independent **PCMCIA** state with the config entry *cfe*.

pcmcia_function_enable(*pf*)
: Provide power to the socket containing the device specified by device function *pf*.

pcmcia_function_disable(*pf*)
: Remove power from the socket containing the device specified by device function *pf*.

pcmcia_io_alloc(*pf*, *start*, *size*, *align*, *pciop*)
Request I/O space for device function *pf* at address *start* of size *size*. Alignment is specified by *align*. A handle for the I/O space is returned in *pciop*.

pcmcia_io_free(*pf*, *pcihp*)
Release I/O space with handle *pcihp* for device function *pf*.

pcmcia_io_map(*pf*, *width*, *pcihp*, *windowp*)
Map device I/O for device function *pf* to the I/O space with handle *pcihp*. The width of data access is specified by *width*. Valid values for the width are:

PCMCIA_WIDTH_AUTO
Use the largest I/O width reported by the device.

PCMCIA_WIDTH_IO8 Force 8-bit I/O width.

PCMCIA_WIDTH_IO16
Force 16-bit I/O width.

A handle for the mapped I/O window is returned in *windowp*.

pcmcia_io_unmap(*pf*, *window*)
Unmap the I/O window *window* for device function *pf*.

pcmcia_mem_alloc(*pf*, *size*, *pcmhp*)
Request memory space for device function *pf* of size *size*. A handle for the memory space is returned in *pcmhp*.

pcmcia_mem_free(*pf*, *pcmhp*)
Release memory space with handle *pcmhp* for device function *pf*.

pcmcia_mem_map(*pf*, *width*, *card_addr*, *size*, *pcmhp*, *offsetp*, *windowp*)
Map device memory for device function *pf* to the memory space with handle *pcmhp*. The address of the device memory starts at *card_addr* and is size *size*. The width of data access is specified by *width*. Valid values for the width are:

PCMCIA_WIDTH_MEM8
Force 8-bit memory width.

PCMCIA_WIDTH_MEM16
Force 16-bit memory width.

A handle for the mapped memory window is returned in *windowp* and a bus-space offset into the memory window is returned in *offsetp*.

pcmcia_mem_unmap(*pf*, *window*)
Unmap the memory window *window* for device function *pf*.

pcmcia_intr_establish(*pf*, *level*, *handler*, *arg*)
Establish an interrupt handler for device function *pf*. The priority of the interrupt is specified by *level*. When the interrupt occurs the function *handler* is called with argument *arg*. The return value is a handle for the interrupt handler. **pcmcia_intr_establish**() returns an opaque handle to an event descriptor if it succeeds, and returns NULL on failure.

pcmcia_intr_disestablish(*pf*, *ih*)
Dis-establish the interrupt handler for device function *pf* with handle *ih*. The handle was returned from **pcmcia_intr_establish**().

pcmcia_cis_read_1(*tuple*, *index*)
Read one byte from tuple *tuple* at index *index* in the CIS.

pcmcia_cis_read_2(*tuple*, *index*)
Read two bytes from tuple *tuple* at index *index* in the CIS.

pcmcia_cis_read_3(*tuple*, *index*)
Read three bytes from tuple *tuple* at index *index* in the CIS.

pcmcia_cis_read_4(*tuple*, *index*)
Read four bytes from tuple *tuple* at index *index* in the CIS.

pcmcia_cis_read_n(*tuple*, *number*, *index*)
Read *n* bytes from tuple *tuple* at index *index* in the CIS.

pcmcia_scan_cis(*dev*, *func*, *arg*)
Scan the CIS for device *dev*. For each tuple in the CIS, function *func* is called with the tuple and the argument *arg*. *func* should return 0 if the tuple it was called with is the one it was looking for, or 1 otherwise.

AUTOCONFIGURATION

During autoconfiguration, a **PCMCIA** driver will receive a pointer to *struct pcmcia_attach_args* describing the device attached to the PCMCIA bus. Drivers match the device using the *manufacturer* and *product* members.

During the driver attach step, drivers will use the pcmcia function *pf*. The driver should traverse the list of config entries searching for a useful configuration. This config entry is passed to **pcmcia_function_init**() to initialise the machine-independent interface. I/O and memory resources should be initialised using **pcmcia_io_alloc**() and **pcmcia_mem_alloc**() using the specified resources in the config entry. These resources can then be mapped into processor bus space using **pcmcia_io_map**() and **pcmcia_mem_map**() respectively. Upon successful allocation of resources, power can be applied to the device with **pcmcia_function_enable**() so that device-specific interrogation can be performed. Finally, power should be removed from the device using **pcmcia_function_disable**().

Since PCMCIA devices support dynamic configuration, drivers should make use of **powerhook_establish**(*9*). Power can be applied and the interrupt handler should be established through this interface.

DMA SUPPORT

PCMCIA devices do not support DMA.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent PCMCIA subsystem can be found. All pathnames are relative to `/usr/src`.

The PCMCIA subsystem itself is implemented within the file `sys/dev/pcmcia/pcmcia.c`. The database of known devices exists within the file `sys/dev/pcmcia/pcmciadevs_data.h` and is generated automatically from the file `sys/dev/pcmcia/pcmciadevs`. New manufacturer and product identifiers should be added to this file. The database can be regenerated using the Makefile `sys/dev/pcmcia/Makefile.pcmciadevs`.

SEE ALSO

`pcic`(4), `pcmcia`(4), `tcic`(4), `autoconf`(9), `bus_dma`(9), `bus_space`(9), `driver`(9)

Personal Computer Memory Card International Association (PCMCIA), *PC Card 95 Standard*, 1995.

HISTORY

The machine-independent PCMCIA subsystem appeared in NetBSD 1.3.

NAME

pfil, **pfil_head_register**, **pfil_head_unregister**, **pfil_head_get**, **pfil_hook_get**, **pfil_add_hook**, **pfil_remove_hook**, **pfil_run_hooks** — packet filter interface

SYNOPSIS

```
#include <sys/param.h>
#include <sys/mbuf.h>
#include <net/if.h>
#include <net/pfil.h>

int
pfil_head_register(struct pfil_head *ph);

int
pfil_head_unregister(struct pfil_head *ph);

struct pfil_head *
pfil_head_get(int af, u_long dlt);

struct packet_filter_hook *
pfil_hook_get(int dir, struct pfil_head *ph);

int
pfil_add_hook(int (*func)(), void *arg, int flags, struct pfil_head *ph);

int
pfil_remove_hook(int (*func)(), void *arg, int flags, struct pfil_head *ph);

int
(*func)(void *arg, struct mbuf **mp, struct ifnet *, int dir);

int
pfil_run_hooks(struct pfil_head *ph, struct mbuf **mp, struct ifnet *ifp,
    int dir);
```

DESCRIPTION

The **pfil** framework allows for a specified function to be invoked for every incoming or outgoing packet for a particular network I/O stream. These hooks may be used to implement a firewall or perform packet transformations.

Packet filtering points are registered with **pfil_head_register**(). Filtering points are identified by a key (void *) and a data link type (int) in the *pfil_head* structure. Packet filters use the key and data link type to look up the filtering point with which they register themselves. The key is unique to the filtering point. The data link type is a bpf(4) DLT constant indicating what kind of header is present on the packet at the filtering point. Filtering points may be unregistered with the **pfil_head_unregister**() function.

Packet filters register/unregister themselves with a filtering point with the **pfil_add_hook**() and **pfil_remove_hook**() functions, respectively. The head is looked up using the **pfil_head_get**() function, which takes the key and data link type that the packet filter expects. Filters may provide an argument to be passed to the filter when invoked on a packet.

When a filter is invoked, the packet appears just as if it "came off the wire". That is, all protocol fields are in network byte order. The filter is called with its specified argument, the pointer to the pointer to the mbuf containing the packet, the pointer to the network interface that the packet is traversing, and the direction (PFIL_IN or PFIL_OUT, see also below) that the packet is traveling. The filter may change which mbuf the mbuf ** argument references. The filter returns an errno if the packet processing is to stop, or 0 if the processing is to continue. If the packet processing is to stop, it is the responsibility of the filter to free the

packet.

The *flags* parameter, used in the **pfil_add_hook**() and **pfil_remove_hook**() functions, indicates when the filter should be called. The flags are:

PFIL_IN	call me on incoming packets
PFIL_OUT	call me on outgoing packets
PFIL_ALL	call me on all of the above
PFIL_IFADDR	call me on interface reconfig (mbuf ** is ioctl #)
PFIL_IFNET	call me on interface attach/detach (mbuf ** is either PFIL_IFNET_ATTACH or PFIL_IFNET_DETACH)
PFIL_WAITOK	OK to call malloc with M_WAITOK.

The **pfil** interface is enabled in the kernel via the **PFIL_HOOKS** option.

SEE ALSO

bpf(4)

HISTORY

The **pfil** interface first appeared in NetBSD 1.3. The **pfil** input and output lists were originally implemented as ⟨sys/queue.h⟩ LIST structures; however this was changed in NetBSD 1.4 to TAILQ structures. This change was to allow the input and output filters to be processed in reverse order, to allow the same path to be taken, in or out of the kernel.

The **pfil** interface was changed in 1.4T to accept a 3rd parameter to both **pfil_add_hook**() and **pfil_remove_hook**(), introducing the capability of per-protocol filtering. This was done primarily in order to support filtering of IPv6.

In 1.5K, the **pfil** framework was changed to work with an arbitrary number of filtering points, as well as be less IP-centric.

AUTHORS

The **pfil** interface was designed and implemented by Matthew R. Green, with help from Darren Reed, Jason R. Thorpe and Charles M. Hannum. Darren Reed added support for IPv6 in addition to IPv4. Jason R. Thorpe added support for multiple hooks and other clean up.

BUGS

The current **pfil** implementation will need changes to suit a threaded kernel model.

NAME

physio — initiate I/O on raw devices

SYNOPSIS

```
int
physio((*strategy)(struct buf *),struct buf *bp,dev_t dev,int flags,
    (*minphys)(struct buf *),struct uio *uio);
```

DESCRIPTION

The **physio**() is a helper function typically called from character device read and write routines to start I/O on a user process buffer. It calls back on the provided *strategy* routine one or more times to complete the transfer described by *uio*. The maximum amount of data to transfer with each call to *strategy* is determined by the *minphys* routine. Since *uio* normally describes user space addresses, **physio**() needs to lock the appropriate data area into memory before each transaction with *strategy* (*see* uvm_vslock(9) and uvm_vsunlock(9)). **physio**() always awaits the completion of the entire requested transfer before returning, unless an error condition is detected earlier. In all cases, the buffer passed in *bp* is locked (marked as "busy") for the duration of the entire transfer.

A break-down of the arguments follows:

strategy
: The device strategy routine to call for each chunk of data to initiate device I/O.

bp
: The buffer to use with the strategy routine. The buffer flags will have B_BUSY, B_PHYS, and B_RAW set when passed to the strategy routine. If NULL, a buffer is allocated from a system pool.

dev
: The device number identifying the device to interact with.

flags
: Direction of transfer; the only valid settings are B_READ or B_WRITE.

minphys
: A device specific routine called to determine the maximum transfer size that the device's strategy routine can handle.

uio
: The description of the entire transfer as requested by the user process. Currently, the results of passing a *uio* structure with the 'uio_segflg' set to anything other than UIO_USERSPACE, are undefined.

RETURN VALUES

If successful **physio**() returns 0. EFAULT is returned if the address range described by *uio* is not accessible by the requesting process. **physio**() will return any error resulting from calls to the device strategy routine, by examining the B_ERROR buffer flag and the 'b_error' field. Note that the actual transfer size may be less than requested by *uio* if the device signals an "end of file" condition.

SEE ALSO

read(2), write(2)

NAME

pmap — machine-dependent portion of the virtual memory system

SYNOPSIS

```
#include <sys/param.h>
#include <uvm/uvm_extern.h>

void
pmap_init(void);

void
pmap_virtual_space(vaddr_t *vstartp, vaddr_t *vendp);

vaddr_t
pmap_steal_memory(vsize_t size, vaddr_t *vstartp, vaddr_t *vendp);

pmap_t
pmap_kernel(void);

pmap_t
pmap_create(void);

void
pmap_destroy(pmap_t pmap);

void
pmap_reference(pmap_t pmap);

void
pmap_fork(pmap_t src_map, pmap_t dst_map);

long
pmap_resident_count(pmap_t pmap);

long
pmap_wired_count(pmap_t pmap);

vaddr_t
pmap_growkernel(vaddr_t maxkvaddr);

int
pmap_enter(pmap_t pmap, vaddr_t va, paddr_t pa, vm_prot_t prot, int flags);

void
pmap_remove(pmap_t pmap, vaddr_t sva, vaddr_t eva);

void
pmap_remove_all(pmap_t pmap);

void
pmap_protect(pmap_t pmap, vaddr_t sva, vaddr_t eva, vm_prot_t prot);

void
pmap_unwire(pmap_t pmap, vaddr_t va);

bool
pmap_extract(pmap_t pmap, vaddr_t va, paddr_t *pap);

void
pmap_kenter_pa(vaddr_t va, paddr_t pa, vm_prot_t prot);
```

void
pmap_kremove(*vaddr_t va*, *vsize_t size*);

void
pmap_copy(*pmap_t dst_map*, *pmap_t src_map*, *vaddr_t dst_addr*, *vsize_t len*, *vaddr_t src_addr*);

void
pmap_collect(*pmap_t pmap*);

void
pmap_update(*pmap_t pmap*);

void
pmap_activate(*struct lwp *l*);

void
pmap_deactivate(*struct lwp *l*);

void
pmap_zero_page(*paddr_t pa*);

void
pmap_copy_page(*paddr_t src*, *paddr_t dst*);

void
pmap_page_protect(*struct vm_page *pg*, *vm_prot_t prot*);

bool
pmap_clear_modify(*struct vm_page *pg*);

bool
pmap_clear_reference(*struct vm_page *pg*);

bool
pmap_is_modified(*struct vm_page *pg*);

bool
pmap_is_referenced(*struct vm_page *pg*);

paddr_t
pmap_phys_address(*paddr_t cookie*);

vaddr_t
PMAP_MAP_POOLPAGE(*paddr_t pa*);

paddr_t
PMAP_UNMAP_POOLPAGE(*vaddr_t va*);

void
PMAP_PREFER(*vaddr_t hint*, *vaddr_t *vap*, *vsize_t sz*, *int td*);

DESCRIPTION

The **pmap** module is the machine-dependent portion of the NetBSD virtual memory system uvm(9). The purpose of the **pmap** module is to manage physical address maps, to program the memory management hardware on the system, and perform any cache operations necessary to ensure correct operation of the virtual memory system. The **pmap** module is also responsible for maintaining certain information required by uvm(9).

In order to cope with hardware architectures that make the invalidation of virtual address mappings expensive (e.g., TLB invalidations, TLB shootdown operations for multiple processors), the **pmap** module is allowed to delay mapping invalidation or protection operations until such time as they are actually necessary. The functions that are allowed to delay such actions are **pmap_enter**(), **pmap_remove**(), **pmap_protect**(), **pmap_kenter_pa**(), and **pmap_kremove**(). Callers of these functions must use the **pmap_update**() function to notify the **pmap** module that the mappings need to be made correct. Since the **pmap** module is provided with information as to which processors are using a given physical map, the **pmap** module may use whatever optimizations it has available to reduce the expense of virtual-to-physical mapping synchronization.

HEADER FILES AND DATA STRUCTURES

Machine-dependent code must provide the header file ⟨machine/pmap.h⟩. This file contains the definition of the pmap structure:

```
struct pmap {
        /* Contents defined by pmap implementation. */
};
typedef struct pmap *pmap_t;
```

This header file may also define other data structures that the **pmap** implementation uses.

Note that all prototypes for **pmap** interface functions are provided by the header file ⟨uvm/uvm_pmap.h⟩. It is possible to override this behavior by defining the C pre-processor macro PMAP_EXCLUDE_DECLS. This may be used to add a layer of indirection to **pmap** API calls, for handling different MMU types in a single **pmap** module, for example. If the PMAP_EXCLUDE_DECLS macro is defined, ⟨machine/pmap.h⟩ *must* provide function prototypes in a block like so:

```
#ifdef _KERNEL /* not exposed to user namespace */
__BEGIN_DECLS  /* make safe for C++ */
/* Prototypes go here. */
__END_DECLS
#endif /* _KERNEL */
```

The header file ⟨uvm/uvm_pmap.h⟩ defines a structure for tracking **pmap** statistics (see below). This structure is defined as:

```
struct pmap_statistics {
        long        resident_count; /* number of mapped pages */
        long        wired_count;    /* number of wired pages */
};
```

WIRED MAPPINGS

The **pmap** module is based on the premise that all information contained in the physical maps it manages is redundant. That is, physical map information may be "forgotten" by the **pmap** module in the event that it is necessary to do so; it can be rebuilt by uvm(9) by taking a page fault. There is one exception to this rule: so-called "wired" mappings may not be forgotten. Wired mappings are those for which either no high-level information exists with which to rebuild the mapping, or mappings which are needed by critical sections of code where taking a page fault is unacceptable. Information about which mappings are wired is provided to the **pmap** module when a mapping is established.

MODIFIED/REFERENCED INFORMATION

The **pmap** module is required to keep track of whether or not a page managed by the virtual memory system has been referenced or modified. This information is used by uvm(9) to determine what happens to the page when scanned by the pagedaemon.

Many CPUs provide hardware support for tracking modified/referenced information. However, many CPUs, particularly modern RISC CPUs, do not. On CPUs which lack hardware support for modified/referenced tracking, the **pmap** module must emulate it in software. There are several strategies for doing this, and the best strategy depends on the CPU.

The "referenced" attribute is used by the pagedaemon to determine if a page is "active". Active pages are not candidates for re-use in the page replacement algorithm. Accurate referenced information is not required for correct operation; if supplying referenced information for a page is not feasible, then the **pmap** implementation should always consider the "referenced" attribute to be `false`.

The "modified" attribute is used by the pagedaemon to determine if a page needs to be cleaned (written to backing store; swap space, a regular file, etc.). Accurate modified information *must* be provided by the **pmap** module for correct operation of the virtual memory system.

Note that modified/referenced information is only tracked for pages managed by the virtual memory system (i.e., pages for which a vm_page structure exists). In addition, only "managed" mappings of those pages have modified/referenced tracking. Mappings entered with the **pmap_enter**() function are "managed" mappings. It is possible for "unmanaged" mappings of a page to be created, using the **pmap_kenter_pa**() function. The use of "unmanaged" mappings should be limited to code which may execute in interrupt context (for example, the kernel memory allocator), or to enter mappings for physical addresses which are not managed by the virtual memory system. "Unmanaged" mappings may only be entered into the kernel's virtual address space. This constraint is placed on the callers of the **pmap_kenter_pa**() and **pmap_kremove**() functions so that the **pmap** implementation need not block interrupts when manipulating data structures or holding locks.

Also note that the modified/referenced information must be tracked on a per-page basis; they are not attributes of a mapping, but attributes of a page. Therefore, even after all mappings for a given page have been removed, the modified/referenced information for that page *must* be preserved. The only time the modified/referenced attributes may be cleared is when the virtual memory system explicitly calls the **pmap_clear_modify**() and **pmap_clear_reference**() functions. These functions must also change any internal state necessary to detect the page being modified or referenced again after the modified or referenced state is cleared. (Prior to NetBSD 1.6, **pmap** implementations could get away without this because UVM (and Mach VM before that) always called **pmap_page_protect**() before clearing the modified or referenced state, but UVM has been changed to not do this anymore, so all **pmap** implementations must now handle this.)

STATISTICS

The **pmap** is required to keep statistics as to the number of "resident" pages and the number of "wired" pages.

A "resident" page is one for which a mapping exists. This statistic is used to compute the resident size of a process and enforce resource limits. Only pages (whether managed by the virtual memory system or not) which are mapped into a physical map should be counted in the resident count.

A "wired" page is one for which a wired mapping exists. This statistic is used to enforce resource limits.

Note that it is recommended (though not required) that the **pmap** implementation use the `pmap_statistics` structure in the tracking of **pmap** statistics by placing it inside the `pmap` structure and adjusting the counts when mappings are established, changed, or removed. This avoids potentially expensive data structure traversals when the statistics are queried.

REQUIRED FUNCTIONS

This section describes functions that a **pmap** module must provide to the virtual memory system.

void **pmap_init**(*void*)
This function initializes the **pmap** module. It is called by **uvm_init**() to initialize any data structures that the module needs to manage physical maps.

pmap_t **pmap_kernel**(*void*)
Return a pointer to the pmap structure that maps the kernel virtual address space.

Note that this function may be provided as a C pre-processor macro.

void **pmap_virtual_space**(*vaddr_t *vstartp*, *vaddr_t *vendp*)
The **pmap_virtual_space**() function is called to determine the initial kernel virtual address space beginning and end. These values are used to create the kernel's virtual memory map. The function must set **vstartp* to the first kernel virtual address that will be managed by uvm(9), and must set **vendp* to the last kernel virtual address that will be managed by uvm(9).

If the **pmap_growkernel**() feature is used by a **pmap** implementation, then **vendp* should be set to the maximum kernel virtual address allowed by the implementation. If **pmap_growkernel**() is not used, then **vendp must* be set to the maximum kernel virtual address that can be mapped with the resources currently allocated to map the kernel virtual address space.

pmap_t **pmap_create**(*void*)
Create a physical map and return it to the caller. The reference count on the new map is 1.

void **pmap_destroy**(*pmap_t pmap*)
Drop the reference count on the specified physical map. If the reference count drops to 0, all resources associated with the physical map are released and the physical map destroyed. In the case of a drop-to-0, no mappings will exist in the map. The **pmap** implementation may assert this.

void **pmap_reference**(*pmap_t pmap*)
Increment the reference count on the specified physical map.

long **pmap_resident_count**(*pmap_t pmap*)
Query the "resident pages" statistic for *pmap*.

Note that this function may be provided as a C pre-processor macro.

long **pmap_wired_count**(*pmap_t pmap*)
Query the "wired pages" statistic for *pmap*.

Note that this function may be provided as a C pre-processor macro.

int **pmap_enter**(*pmap_t pmap*, *vaddr_t va*, *paddr_t pa*, *vm_prot_t prot*, *int flags*)
Create a mapping in physical map *pmap* for the physical address *pa* at the virtual address *va* with protection specified by bits in *prot*:

VM_PROT_READ	The mapping must allow reading.
VM_PROT_WRITE	The mapping must allow writing.
VM_PROT_EXECUTE	The page mapped contains instructions that will be executed by the processor.

The *flags* argument contains protection bits (the same bits as used in the *prot* argument) indicating the type of access that caused the mapping to be created. This information may be used to seed modified/referenced information for the page being mapped, possibly avoiding redundant faults on platforms that track modified/referenced information in

software. Other information provided by *flags*:

PMAP_WIRED The mapping being created is a wired mapping.

PMAP_CANFAIL The call to **pmap_enter**() is allowed to fail. If this flag is *not* set, and the **pmap_enter**() call is unable to create the mapping, perhaps due to insufficient resources, the **pmap** module must panic.

The access type provided in the *flags* argument will never exceed the protection specified by *prot*. The **pmap** implementation may assert this. Note that on systems that do not provide hardware support for tracking modified/referenced information, modified/referenced information for the page *must* be seeded with the access type provided in *flags* if the PMAP_WIRED flag is set. This is to prevent a fault for the purpose of tracking modified/referenced information from occurring while the system is in a critical section where a fault would be unacceptable.

Note that **pmap_enter**() is sometimes called to enter a mapping at a virtual address for which a mapping already exists. In this situation, the implementation must take whatever action is necessary to invalidate the previous mapping before entering the new one.

Also note that **pmap_enter**() is sometimes called to change the protection for a pre-existing mapping, or to change the "wired" attribute for a pre-existing mapping.

The **pmap_enter**() function returns 0 on success or an error code indicating the mode of failure.

void **pmap_remove**(*pmap_t pmap*, *vaddr_t sva*, *vaddr_t eva*)
Remove mappings from the virtual address range *sva* to *eva* from the specified physical map.

void **pmap_remove_all**(*pmap_t pmap*)
This function is a hint to the **pmap** implementation that all entries in *pmap* will be removed before any more entries are entered. Following this call, there will be **pmap_remove**() calls resulting in every mapping being removed, followed by either **pmap_destroy**() or **pmap_update**(). No other **pmap** interfaces which take *pmap* as an argument will be called during this process. Other interfaces which might need to access *pmap* (such as **pmap_page_protect**()) are permitted during this process.

The **pmap** implementation is free to either remove all the **pmap**'s mappings immediately in **pmap_remove_all**(), or to use the knowledge of the upcoming **pmap_remove**() calls to optimize the removals (or to just ignore this call).

void **pmap_protect**(*pmap_t pmap*, *vaddr_t sva*, *vaddr_t eva*, *vm_prot_t prot*)
Set the protection of the mappings in the virtual address range *sva* to *eva* in the specified physical map.

void **pmap_unwire**(*pmap_t pmap*, *vaddr_t va*)
Clear the "wired" attribute on the mapping for virtual address *va*.

bool **pmap_extract**(*pmap_t pmap*, *vaddr_t va*, *paddr_t *pap*)
This function extracts a mapping from the specified physical map. It serves two purposes: to determine if a mapping exists for the specified virtual address, and to determine what physical address is mapped at the specified virtual address. The **pmap_extract**() should return the physical address for any kernel-accessible address, including KSEG-style direct-mapped kernel addresses.

The **pmap_extract**() function returns `false` if a mapping for *va* does not exist. Otherwise, it returns `true` and places the physical address mapped at *va* into **pap* if the *pap* argument is non-NULL.

void **pmap_kenter_pa**(*vaddr_t va*, *paddr_t pa*, *vm_prot_t prot*)
Enter an "unmanaged" mapping for physical address *pa* at virtual address *va* with protection *prot* into the kernel physical map. Mappings of this type are always "wired", and are unaffected by routines that alter the protection of pages (such as **pmap_page_protect**()). Such mappings are also not included in the gathering of modified/referenced information about a page. Mappings entered with **pmap_kenter_pa**() by machine-independent code *must not* have execute permission, as the data structures required to track execute permission of a page may not be available to **pmap_kenter_pa**(). Machine-independent code is not allowed to enter a mapping with **pmap_kenter_pa**() at a virtual address for which a valid mapping already exists. Mappings created with **pmap_kenter_pa**() may be removed only with a call to **pmap_kremove**().

Note that **pmap_kenter_pa**() must be safe for use in interrupt context. **splvm**() blocks interrupts that might cause **pmap_kenter_pa**() to be called.

void **pmap_kremove**(*vaddr_t va*, *vsize_t size*)
Remove all mappings starting at virtual address *va* for *size* bytes from the kernel physical map. All mappings that are removed must be the "unmanaged" type created with **pmap_kenter_pa**(). The implementation may assert this.

void **pmap_copy**(*pmap_t dst_map*, *pmap_t src_map*, *vaddr_t dst_addr*, *vsize_t len*, *vaddr_t src_addr*)
This function copies the mappings starting at *src_addr* in *src_map* for *len* bytes into *dst_map* starting at *dst_addr*.

Note that while this function is required to be provided by a **pmap** implementation, it is not actually required to do anything. **pmap_copy**() is merely advisory (it is used in the `fork`(2) path to "pre-fault" the child's address space).

void **pmap_collect**(*pmap_t pmap*)
This function is called just before a process is swapped out to allow the **pmap** module to release resources used to map the process's address space. The implementation may choose to remove physical mappings in order to free for example page tables back to the system. Note, however, that wired mappings must *not* be removed when **pmap_collect**() is called.

Note that while this function is required to be provided by a **pmap** implementation, it is not actually required to do anything. **pmap_collect**() is merely advisory. It is recommended, however, that **pmap_collect**() be fully implemented by a **pmap** implementation.

void **pmap_update**(*pmap_t pmap*)
This function is used to inform the **pmap** module that all physical mappings, for the specified pmap, must now be correct. That is, all delayed virtual-to-physical mappings updates (such as TLB invalidation or address space identifier updates) must be completed. This routine must be used after calls to **pmap_enter**(), **pmap_remove**(), **pmap_protect**(), **pmap_kenter_pa**(), and **pmap_kremove**() in order to ensure correct operation of the virtual memory system.

If a **pmap** implementation does not delay virtual-to-physical mapping updates, **pmap_update**() has no operation. In this case, the call may be deleted using a C pre-pro-

cessor macro in ⟨machine/pmap.h⟩.

void **pmap_activate**(*struct lwp *l*)
: Activate the physical map used by the process behind lwp *l*. This is called by the virtual memory system when the virtual memory context for a process is changed, and is also often used by machine-dependent context switch code to program the memory management hardware with the process's page table base, etc. Note that **pmap_activate**() may not always be called when *l* is the current lwp. **pmap_activate**() must be able to handle this scenario.

void **pmap_deactivate**(*struct lwp *l*)
: Deactivate the physical map used by the process behind lwp *l*. It is generally used in conjunction with **pmap_activate**(). Like **pmap_activate**(), **pmap_deactivate**() may not always be called when *l* is the current lwp.

void **pmap_zero_page**(*paddr_t pa*)
: Zero the PAGE_SIZE sized region starting at physical address *pa*. The **pmap** implementation must take whatever steps are necessary to map the page to a kernel-accessible address and zero the page. It is suggested that implementations use an optimized zeroing algorithm, as the performance of this function directly impacts page fault performance. The implementation may assume that the region is PAGE_SIZE aligned and exactly PAGE_SIZE bytes in length.

  Note that the cache configuration of the platform should also be considered in the implementation of **pmap_zero_page**(). For example, on systems with a physically-addressed cache, the cache load caused by zeroing the page will not be wasted, as the zeroing is usually done on-demand. However, on systems with a virtually-addressed cached, the cache load caused by zeroing the page *will* be wasted, as the page will be mapped at a virtual address which is different from that used to zero the page. In the virtually-addressed cache case, care should also be taken to avoid cache alias problems.

void **pmap_copy_page**(*paddr_t src*, *paddr_t dst*)
: Copy the PAGE_SIZE sized region starting at physical address *src* to the same sized region starting at physical address *dst*. The **pmap** implementation must take whatever steps are necessary to map the source and destination pages to a kernel-accessible address and perform the copy. It is suggested that implementations use an optimized copy algorithm, as the performance of this function directly impacts page fault performance. The implementation may assume that both regions are PAGE_SIZE aligned and exactly PAGE_SIZE bytes in length.

  The same cache considerations that apply to **pmap_zero_page**() apply to **pmap_copy_page**().

void **pmap_page_protect**(*struct vm_page *pg*, *vm_prot_t prot*)
: Lower the permissions for all mappings of the page *pg* to *prot*. This function is used by the virtual memory system to implement copy-on-write (called with VM_PROT_READ set in *prot*) and to revoke all mappings when cleaning a page (called with no bits set in *prot*). Access permissions must never be added to a page as a result of this call.

bool **pmap_clear_modify**(*struct vm_page *pg*)
: Clear the "modified" attribute on the page *pg*.

  The **pmap_clear_modify**() function returns `true` or `false` indicating whether or not the "modified" attribute was set on the page before it was cleared.

Note that this function may be provided as a C pre-processor macro.

bool **pmap_clear_reference**(*struct vm_page *pg*)
Clear the "referenced" attribute on the page *pg*.

The **pmap_clear_reference**() function returns `true` or `false` indicating whether or not the "referenced" attribute was set on the page before it was cleared.

Note that this function may be provided as a C pre-processor macro.

bool **pmap_is_modified**(*struct vm_page *pg*)
Test whether or not the "modified" attribute is set on page *pg*.

Note that this function may be provided as a C pre-processor macro.

bool **pmap_is_referenced**(*struct vm_page *pg*)
Test whether or not the "referenced" attribute is set on page *pg*.

Note that this function may be provided as a C pre-processor macro.

paddr_t **pmap_phys_address**(*paddr_t cookie*)
Convert a cookie returned by a device **mmap**() function into a physical address. This function is provided to accommodate systems which have physical address spaces larger than can be directly addressed by the platform's *paddr_t* type. The existence of this function is highly dubious, and it is expected that this function will be removed from the **pmap** API in a future release of NetBSD.

Note that this function may be provided as a C pre-processor macro.

OPTIONAL FUNCTIONS

This section describes several optional functions in the **pmap** API.

vaddr_t **pmap_steal_memory**(*vsize_t size*, *vaddr_t *vstartp*, *vaddr_t *vendp*)
This function is a bootstrap memory allocator, which may be provided as an alternative to the bootstrap memory allocator used within uvm(9) itself. It is particularly useful on systems which provide for example a direct-mapped memory segment. This function works by stealing pages from the (to be) managed memory pool, which has already been provided to uvm(9) in the vm_physmem[] array. The pages are then mapped, or otherwise made accessible to the kernel, in a machine-dependent way. The memory must be zeroed by **pmap_steal_memory**(). Note that memory allocated with **pmap_steal_memory**() will never be freed, and mappings made by **pmap_steal_memory**() must never be "forgotten".

Note that **pmap_steal_memory**() should not be used as a general-purpose early-startup memory allocation routine. It is intended to be used only by the **uvm_pageboot_alloc**() routine and its supporting routines. If you need to allocate memory before the virtual memory system is initialized, use **uvm_pageboot_alloc**(). See uvm(9) for more information.

The **pmap_steal_memory**() function returns the kernel-accessible address of the allocated memory. If no memory can be allocated, or if allocated memory cannot be mapped, the function must panic.

If the **pmap_steal_memory**() function uses address space from the range provided to uvm(9) by the **pmap_virtual_space**() call, then **pmap_steal_memory**() must adjust *vstartp* and *vendp* upon return.

The **pmap_steal_memory**() function is enabled by defining the C pre-processor macro PMAP_STEAL_MEMORY in ⟨machine/pmap.h⟩.

vaddr_t **pmap_growkernel**(*vaddr_t maxkvaddr*)

Management of the kernel virtual address space is complicated by the fact that it is not always safe to wait for resources with which to map a kernel virtual address. However, it is not always desirable to pre-allocate all resources necessary to map the entire kernel virtual address space.

The **pmap_growkernel**() interface is designed to help alleviate this problem. The virtual memory startup code may choose to allocate an initial set of mapping resources (e.g., page tables) and set an internal variable indicating how much kernel virtual address space can be mapped using those initial resources. Then, when the virtual memory system wishes to map something at an address beyond that initial limit, it calls **pmap_growkernel**() to pre-allocate more sources with which to create the mapping. Note that once additional kernel virtual address space mapping resources have been allocated, they should not be freed; it is likely they will be needed again.

The **pmap_growkernel**() function returns the new maximum kernel virtual address that can be mapped with the resources it has available. If new resources cannot be allocated, **pmap_growkernel**() must panic.

The **pmap_growkernel**() function is enabled by defining the C pre-processor macro PMAP_GROWKERNEL in ⟨machine/pmap.h⟩.

void **pmap_fork**(*pmap_t src_map*, *pmap_t dst_map*)

Some **pmap** implementations may need to keep track of other information not directly related to the virtual address space. For example, on the i386 port, the Local Descriptor Table state of a process is associated with the pmap (this is due to the fact that applications manipulate the Local Descriptor Table directly expect it to be logically associated with the virtual memory state of the process).

The **pmap_fork**() function is provided as a way to associate information from *src_map* with *dst_map* when a vmspace is forked. **pmap_fork**() is called from **uvmspace_fork**().

The **pmap_fork**() function is enabled by defining the C pre-processor macro PMAP_FORK in ⟨machine/pmap.h⟩.

vaddr_t **PMAP_MAP_POOLPAGE**(*paddr_t pa*)

This function is used by the pool(9) memory pool manager. Pools allocate backing pages one at a time. This is provided as a means to use hardware features such as a direct-mapped memory segment to map the pages used by the pool(9) allocator. This can lead to better performance by e.g. reducing TLB contention.

PMAP_MAP_POOLPAGE() returns the kernel-accessible address of the page being mapped. It must always succeed.

The use of **PMAP_MAP_POOLPAGE**() is enabled by defining it as a C pre-processor macro in ⟨machine/pmap.h⟩. If **PMAP_MAP_POOLPAGE**() is defined, **PMAP_UNMAP_POOLPAGE**() must also be defined.

The following is an example of how to define **PMAP_MAP_POOLPAGE**():

```
#define PMAP_MAP_POOLPAGE(pa)   MIPS_PHYS_TO_KSEG0((pa))
```

This takes the physical address of a page and returns the KSEG0 address of that page on a MIPS processor.

paddr_t **PMAP_UNMAP_POOLPAGE**(*vaddr_t va*)
This function is the inverse of **PMAP_MAP_POOLPAGE**().

PMAP_UNMAP_POOLPAGE() returns the physical address of the page corresponding to the provided kernel-accessible address.

The use of **PMAP_UNMAP_POOLPAGE**() is enabled by defining it as a C pre-processor macro in ⟨machine/pmap.h⟩. If **PMAP_UNMAP_POOLPAGE**() is defined, **PMAP_MAP_POOLPAGE**() must also be defined.

The following is an example of how to define **PMAP_UNMAP_POOLPAGE**():

```
#define PMAP_UNMAP_POOLPAGE(pa) MIPS_KSEG0_TO_PHYS((va))
```

This takes the KSEG0 address of a previously-mapped pool page and returns the physical address of that page on a MIPS processor.

void **PMAP_PREFER**(*vaddr_t hint*, *vaddr_t *vap*, *vsize_t sz*, *int td*)
This function is used by uvm_map(9) to adjust a virtual address being allocated in order to avoid cache alias problems. If necessary, the virtual address pointed by *vap* will be advanced. *hint* is an object offset which will be mapped into the resulting virtual address, and *sz* is size of the object. *td* indicates if the machine dependent pmap uses the topdown VM.

The use of **PMAP_PREFER**() is enabled by defining it as a C pre-processor macro in ⟨machine/pmap.h⟩.

void **pmap_procwr**(*struct proc *p*, *vaddr_t va*, *vsize_t size*)
Synchronize CPU instruction caches of the specified range. The address space is designated by *p*. This function is typically used to flush instruction caches after code modification.

The use of **pmap_procwr**() is enabled by defining a C pre-processor macro PMAP_NEED_PROCWR in ⟨machine/pmap.h⟩.

SEE ALSO

uvm(9)

HISTORY

The **pmap** module was originally part of the design of the virtual memory system in the Mach Operating System. The goal was to provide a clean separation between the machine-independent and the machine-dependent portions of the virtual memory system, in stark contrast to the original 3BSD virtual memory system, which was specific to the VAX.

Between 4.3BSD and 4.4BSD, the Mach virtual memory system, including the **pmap** API, was ported to BSD and included in the 4.4BSD release.

NetBSD inherited the BSD version of the Mach virtual memory system. NetBSD 1.4 was the first NetBSD release with the new uvm(9) virtual memory system, which included several changes to the **pmap** API. Since the introduction of uvm(9), the **pmap** API has evolved further.

AUTHORS

The original Mach VAX **pmap** module was written by Avadis Tevanian, Jr. and
Michael Wayne Young.

Mike Hibler did the integration of the Mach virtual memory system into 4.4BSD and implemented a **pmap** module for the Motorola 68020+68851/68030/68040.

The **pmap** API as it exists in NetBSD is derived from 4.4BSD, and has been modified by
Chuck Cranor,
Charles M. Hannum,
Chuck Silvers,
Wolfgang Solfrank,
Bill Sommerfeld, and
Jason R. Thorpe.

The author of this document is
Jason R. Thorpe ⟨thorpej@NetBSD.org⟩.

BUGS

The use and definition of **pmap_activate**() and **pmap_deactivate**() needs to be reexamined.

The use of **pmap_copy**() needs to be reexamined. Empirical evidence suggests that performance of the system suffers when **pmap_copy**() actually performs its defined function. This is largely due to the fact that the copy of the virtual-to-physical mappings is wasted if the process calls `execve`(2) after `fork`(2). For this reason, it is recommended that **pmap** implementations leave the body of the **pmap_copy**() function empty for now.

NAME

pmatch — performs pattern matching on strings

SYNOPSIS

```
#include <sys/systm.h>

int
pmatch(const char *string, const char *pattern, const char **estr);
```

DESCRIPTION

Extract substring matching *pattern* from *string*. If not NULL, *estr* points to the end of the longest exact or substring match.

pmatch() uses the following metacharacters:

`?` match any single character.

`*` match any character 0 or more times.

`[` define a range of characters that will match. The range is defined by 2 characters separated by a '`-`'. The range definition has to end with a '`]`'. A '`^`' following the '`[`' will negate the range.

RETURN VALUES

pmatch() will return 2 for an exact match, 1 for a substring match, 0 for no match and −1 if an error occurs.

NAME

pmc, **pmc_get_num_counters**, **pmc_get_counter_type**, **pmc_save_context**, **pmc_restore_context**, **pmc_enable_counter**, **pmc_disable_counter**, **pmc_counter_isrunning**, **pmc_counter_isconfigured**, **pmc_configure_counter**, **pmc_get_counter_value**, **pmc_accumulate**, **pmc_alloc_kernel_counter**, **pmc_free_kernel_counter**, **pmc_start_profiling**, **pmc_stop_profiling**, **PMC_ENABLED** — Hardware Performance Monitoring Interface

SYNOPSIS

```
#include <sys/pmc.h>

int
pmc_get_num_counters(void);

int
pmc_get_counter_type(int ctr);

void
pmc_save_context(struct lwp *l);

void
pmc_restore_context(struct lwp *l);

int
pmc_enable_counter(struct lwp *l, int ctr);

int
pmc_disable_counter(struct lwp *l, int ctr);

int
pmc_counter_isrunning(struct lwp *l, int ctr);

int
pmc_counter_isconfigured(struct lwp *l, int ctr);

int
pmc_configure_counter(struct lwp *l, int ctr, struct pmc_counter_cfg *cfg);

int
pmc_get_counter_value(struct lwp *l, int ctr, int flags, uint64_t *pval);

int
pmc_accumulate(struct lwp *l_parent, struct lwp *l_exiting);

int
pmc_alloc_kernel_counter(int ctr, struct pmc_counter_cfg *cfg);

int
pmc_free_kernel_counter(int ctr);

int
pmc_start_profiling(int ctr, struct pmc_counter_cfg *cfg);

int
pmc_stop_profiling(int ctr);

int
PMC_ENABLED(struct lwp *l);
```

DESCRIPTION

Provides a machine-independent interface to the hardware performance counters which are available on several CPU families. The capabilities of these counters vary from CPU to CPU, but they basically count hardware events such as data cache hits or misses, branches taken, branched mispredicted, and so forth. Some can interrupt the processor when a certain threshold has been reached. Some can count events in user space and kernel space independently.

The **pmc** interface is intended to allow monitoring from within the kernel as well as monitoring of userland applications. If the hardware can interrupt the CPU in a specific implementation, then it may also be used as a profiling source instead of the clock.

NOTES

All function calls in this interface may be defined as `cpp`(1) macros. If any function is not implemented as a macro, its prototype must be defined by the port-specific header ⟨`machine/pmc.h`⟩.

Counters are numbered from 0 to `N-1` where `N` is the number of counters available on the system (see **`pmc_get_num_counters`**() below).

Upon a process fork, implementations must

- Zero performance counters for the new process, and
- Inherit any enabled performance counters.

DATA TYPES

Each implementation must specify two new types:

`pmc_evid_t` An integer type which can contain the event IDs for a given processor.

`pmc_ctr_t` An integer type defining the value which may be contained in a given counter register.

Counters are configured with the *`struct pmc_counter_cfg`*. This structure is defined as

```
struct pmc_counter_cfg {
        pmc_evid_t      event_id;
        pmc_ctr_t       reset_value;
        uint32_t        flags;
};
```

`flags` are currently unused.

FUNCTIONS

`pmc_get_num_counters`(*`void`*)

Returns the number of counters present on the current system. Valid values for *`ctr`* in the interface entry points below are from zero to one less than the return value from this function.

`pmc_get_counter_type`(*`int ctr`*)

Returns an implementation-dependent type describing the specified counter. If *`ctr`* is specified as −1, returns a machine-dependent type describing the CPU or counter configuration. For example, on an ia32 architecture, it may distinguish between 586-, 686-, and K7-style counters.

`pmc_save_context`(*`struct lwp *l`*)

Saves the PMC context for the current process. This is called just before `cpu_switch`(9). If there is kernel PMC state, it must be maintained across this call.

pmc_restore_context(*struct lwp *l*)
Restores the PMC context for the current process. This is called just after cpu_switch(9) returns. If there is kernel PMC state, it must be maintained across this call.

pmc_enable_counter(*struct lwp *l*, *int ctr*)
Enables counter *ctr* for the specified process. The counter should have already been configured with a call to **pmc_configure_counter**(). This starts the counter running if it is not already started and enables any interrupts, as appropriate.

pmc_disable_counter(*struct lwp *l*, *int ctr*)
Disables counter *ctr* for the specified process. This stops the counter from running, and disables any interrupts, as appropriate.

pmc_counter_isrunning(*struct lwp *l*, *int ctr*)
Returns non-zero if the specified counter in the specified process is running or if the counter is running in the kernel.

pmc_counter_isconfigured(*struct lwp *l*, *int ctr*)
Returns non-zero if the specified counter in the specified process is configured or if the counter is in use by the kernel.

pmc_configure_counter(*struct lwp *l*, *int ctr*, *struct pmc_counter_cfg *cfg*)
Configures counter *ctr* according to the configuration information stored in *cfg*.

pmc_get_counter_value(*struct lwp *l*, *int ctr*, *int flags*, *uint64_t *pval*)
Returns the value of counter *ctr* in the space pointed to by *pval*. The only recognized flag is *PMC_VALUE_FLAGS_CHILDREN* which specifies that the returned counts should be accumulated values for any exited child processes.

pmc_accumulate(*struct lwp *l_parent*, *struct lwp *l_exiting*)
Accumulates any counter data from the exiting process *p_exiting* into the counters for the parent process *p_parent*.

pmc_alloc_kernel_counter(*int ctr*, *struct pmc_counter_cfg *cfg*)
Allocates counter *ctr* for use by the kernel and configures it with *cfg*.

pmc_free_kernel_counter(*int ctr*)
Returns counter *ctr* to the available pool of counters that may be used by processes.

pmc_start_profiling(*int ctr*, *struct pmc_counter_cfg *cfg*)
Allocates counter *ctr* for use by the kernel for profiling and configures it with *cfg*.

pmc_stop_profiling(*int ctr*)
Stops profiling with counter *ctr*.

PMC_ENABLED(*struct lwp *l*)
Returns non-zero if the given process or the kernel is using the PMC at all.

SEE ALSO

pmc(1), pmc_control(2), pmc_get_info(2)

HISTORY

The **pmc** interface appeared in NetBSD 2.0.

AUTHORS

The **pmc** interface was designed and implemented by Allen Briggs for Wasabi Systems, Inc. Additional input on the **pmc** design was provided by Jason R. Thorpe.

NAME

PMF, **pmf_device_register**, **pmf_device_deregister**, **pmf_device_suspend**, **pmf_device_resume**, **pmf_device_recursive_suspend**, **pmf_device_recursive_resume**, **pmf_device_resume_subtree**, **pmf_class_network_register**, **pmf_class_input_register**, **pmf_class_display_register**, **pmf_system_suspend**, **pmf_system_resume**, **pmf_system_shutdown**, **pmf_event_register**, **pmf_event_deregister**, **pmf_event_inject**, **pmf_set_platform**, **pmf_get_platform** — power management and inter-driver messaging framework

SYNOPSIS

```
#include <sys/device.h>

bool
pmf_device_register(device_t dev, bool (*suspend)(device_t dev),
    bool (*resume)(device_t dev));

pmf_device_register1(device_t dev, bool (*suspend)(device_t dev),
    bool (*resume)(device_t dev),
    bool (*shutdown)(device_t dev, int how));

void
pmf_device_deregister(device_t dev);

bool
pmf_device_suspend(device_t dev);

bool
pmf_device_resume(device_t dev);

bool
pmf_device_recursive_suspend(device_t dev);

bool
pmf_device_recursive_resume(device_t dev);

bool
pmf_device_resume_subtree(device_t dev);

void
pmf_class_network_register(device_t dev, struct ifnet *ifp);

bool
pmf_class_input_register(device_t dev);

bool
pmf_class_display_register(device_t dev);

bool
pmf_system_suspend(void);

bool
pmf_system_resume(void);

void
pmf_system_shutdown(int);

bool
pmf_event_register(device_t dev, pmf_generic_event_t ev,
```

*void (*handler)(device_t dev*, *bool global*);

void
pmf_event_deregister(*device_t dev*, *pmf_generic_event_t ev*,
*void (*handler)(device_t dev)*, *bool global*);

bool
pmf_event_inject(*device_t dev*, *pmf_generic_event_t ev*);

bool
pmf_set_platform(*const char *key*, *const char *value*);

*const char **
pmf_get_platform(*const char *key*);

DESCRIPTION

The machine-independent **PMF** framework provides power management and inter-driver messaging support for device drivers.

DATA TYPES

Drivers for devices implementing **PMF** may make use of the following data type:

pmf_generic_event_t

A device driver can register as a listener for specific events, or inject events into the message queue. The following message types are defined:

```
PMFE_DISPLAY_ON
PMFE_DISPLAY_REDUCED
PMFE_DISPLAY_STANDBY
PMFE_DISPLAY_SUSPEND
PMFE_DISPLAY_OFF
PMFE_DISPLAY_BRIGHTNESS_UP
PMFE_DISPLAY_BRIGHTNESS_DOWN
PMFE_AUDIO_VOLUME_DOWN
PMFE_AUDIO_VOLUME_TOGGLE
PMFE_CHASSIS_LID_CLOSE
PMFE_CHASSIS_LID_OPEN
```

FUNCTIONS

pmf_device_register(*dev*, *suspend*, *resume*)

Register a device with the power management framework. If either *suspend* or *resume* is `NULL` then it is assumed that device state does not need to be captured and resumed on a power transition. Bus and class-level power management will still be performed. Returns `false` if there was an error.

pmf_device_register1(*dev*, *suspend*, *resume*, *shutdown*)

Like pmf_device_register, but additionally registers a shutdown handler.

pmf_device_deregister(*dev*)

Deregister a device with the power management framework.

pmf_device_suspend(*dev*)

Suspend a device by first calling the class suspend handler, followed by the driver suspend handler, and finally the bus suspend handler.

pmf_device_resume(*dev*)
Resume a device by first calling the bus resume handler, followed by the driver resume handler, and finally the class resume handler.

pmf_device_recursive_suspend(*dev*)
As **pmf_device_suspend**(), but ensures that all child devices of *dev* are suspended.

pmf_device_recursive_resume(*dev*)
As **pmf_device_resume**(), but ensures that all parent devices of *dev* are resumed.

pmf_device_resume_subtree(*dev*)
As **pmf_device_resume**(), but ensures that all child devices of *dev* are resumed.

pmf_class_network_register(*dev*, *ifp*)
Register a device with the power management framework as a network-class device.

pmf_class_input_register(*dev*)
Register a device with the power management framework as an input-class device.

pmf_class_display_register(*dev*)
Register a device with the power management framework as a display-class device.

pmf_system_suspend(*void*)
Suspend all attached devices. Devices are suspended by traversing the autoconfiguration tree beginning with the leaf nodes. This function will fail if any attached drivers do not support the power management framework.

pmf_system_resume(*void*)
Resume all attached devices. Devices are resumed by traversing the autoconfiguration tree beginning with devices that do not have a parent. This function will fail if any attached drivers do not support the power management framework.

pmf_system_shutdown(*int*)
Shutdown all attached devices. Devices are shut down by traversing the autoconfiguration tree beginning with the leaf nodes. The integer argument is passed to the driver shutdown functions. It should contain the reboot(2) "howto" argument. This function ignores the presence of attached drivers that do not support the power management framework.

pmf_event_register(*dev*, *ev*, *handler*, *global*)
Register the callback *handler* to be called whenever an *ev* event is triggered. If *global* is `true`, *handler* accepts anonymous events from **pmf_event_inject**().

pmf_event_deregister(*dev*, *ev*, *handler*, *global*)
Deregister the callback previously registered with **pmf_event_register**().

pmf_event_inject(*dev*, *ev*)
Inject an inter-driver message into the message queue. If *dev* is `NULL`, the event is considered to be anonymous and one or more drivers may handle this event, otherwise the event is delivered directly to the callback registered by *dev*.

pmf_set_platform(*key*, *value*)
Insert a name-value pair into the platform information database.

pmf_get_platform(*key*)
Retrieve the value for *key* from the platform information database. Returns `NULL` if the key is not present.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the power management framework can be found. All pathnames are relative to `/usr/src`.

The power management framework is implemented within the files `sys/sys/pmf.h`, `sys/sys/device.h`, `sys/kern/kern_pmf.c`, and `sys/kern/subr_autoconf.c`.

SEE ALSO

`autoconf`(9), `driver`(9)

HISTORY

The **`PMF`** framework appeared in NetBSD 5.0.

AUTHORS

Jared D. McNeill ⟨jmcneill@NetBSD.org⟩
Joerg Sonnenberger ⟨joerg@NetBSD.org⟩

NAME

pool_init, **pool_destroy**, **pool_get**, **pool_put**, **pool_prime**, **pool_sethiwat**, **pool_setlowat** — resource-pool manager

SYNOPSIS

```
#include <sys/pool.h>

void
pool_init(struct pool *pp, size_t size, u_int align, u_int align_offset,
    int flags, const char *wchan, struct pool_allocator *palloc, int ipl);

void
pool_destroy(struct pool *pp);

void *
pool_get(struct pool *pp, int flags);

void
pool_put(struct pool *pp, void *item);

int
pool_prime(struct pool *pp, int nitems);

void
pool_sethiwat(struct pool *pp, int n);

void
pool_setlowat(struct pool *pp, int n);
```

DESCRIPTION

These utility routines provide management of pools of fixed-sized areas of memory. Resource pools set aside an amount of memory for exclusive use by the resource pool owner. This can be used by applications to guarantee the availability of a minimum amount of memory needed to continue operation independent of the memory resources currently available from the system-wide memory allocator (malloc(9)).

INITIALIZING A POOL

The function **pool_init**() initializes a resource pool. The arguments are:

pp — The handle identifying the pool resource instance.

size — Specifies the size of the memory items managed by the pool.

align — Specifies the memory address alignment of the items returned by **pool_get**(). This argument must be a power of two. If zero, the alignment defaults to an architecture-specific natural alignment.

align_offset — The offset within an item to which the *align* parameter applies.

flags — Should be set to zero or PR_NOTOUCH. If PR_NOTOUCH is given, free items are never used to keep internal state so that the pool can be used for non memory backed objects.

wchan — The 'wait channel' passed on to cv_wait(9) if **pool_get**() must wait for items to be returned to the pool.

palloc — Can be set to NULL or pool_allocator_kmem, in which case the default kernel memory allocator will be used. It can also be set to pool_allocator_nointr when the pool will never be accessed from interrupt context.

ipl Specifies an interrupt priority level that will block all interrupt handlers that could potentially access the pool.

The **POOL_INIT**() macro can be used to both declare and initialize a resource pool. The **POOL_INIT**() macro has the same arguments as the **pool_init**() function and the resource pool will be initialized automatically during system startup.

DESTROYING A POOL

The function **pool_destroy**() destroys a resource pool. It takes a single argument *pp* identifying the pool resource instance.

ALLOCATING ITEMS FROM A POOL

pool_get() allocates an item from the pool and returns a pointer to it. The arguments are:

pp The handle identifying the pool resource instance.

flags The flags can be used to define behaviour in case the pooled resources are depleted. If no resources are available and PR_NOWAIT is given, **pool_get**() returns NULL. If PR_WAITOK is given and allocation is attempted with no resources available, the function will sleep until items are returned to the pool. If both PR_LIMITFAIL and PR_WAITOK are specified, and the pool has reached its hard limit, **pool_get**() will return NULL without waiting, allowing the caller to do its own garbage collection; however, it will still wait if the pool is not yet at its hard limit.

RETURNING ITEMS TO A POOL

pool_put() returns the pool item pointed at by *item* to the resource pool identified by the pool handle *pp*. If the number of available items in the pool exceeds the maximum pool size set by **pool_sethiwat**() and there are no outstanding requests for pool items, the excess items will be returned to the system. The arguments to **pool_put**() are:

pp The handle identifying the pool resource instance.

item A pointer to a pool item previously obtained by **pool_get**().

PRIMING A POOL

pool_prime() adds items to the pool. Storage space for the items is allocated by using the page allocation routine specified to **pool_create**().

The arguments to **pool_prime**() are:

pp The handle identifying the pool resource instance.

nitems The number of items to add to the pool.

This function may return ENOMEM in case the requested number of items could not be allocated. Otherwise, the return value is 0.

SETTING POOL RESOURCE WATERMARKS

A pool will attempt to increase its resource usage to keep up with the demand for its items. Conversely, it will return unused memory to the system should the number of accumulated unused items in the pool exceed a programmable limit. The limits for the minimum and maximum number of items which a pool should keep at hand are known as the high and low **watermarks**. The functions **pool_sethiwat**() and **pool_setlowat**() set a pool's high and low watermarks, respectively.

pool_sethiwat()

pp The handle identifying the pool resource instance.

n The maximum number of items to keep in the pool. As items are returned and the total number of pages in the pool is larger than the maximum set by this function, any completely unused pages are released immediately. If this function is not used to specify a maximum number of items, the pages will remain associated with the pool until the system runs low on memory, at which point the VM system will try to reclaim unused pages.

pool_setlowat()

pp The handle identifying the pool resource instance.

n The minimum number of items to keep in the pool. The number pages in the pool will not decrease below the required value to accommodate the minimum number of items specified by this function. Unlike **pool_prime**(), this function does not allocate the necessary memory up-front.

POTENTIAL PITFALLS

Note that undefined behaviour results when mixing the storage providing methods supported by the pool resource routines.

The pool resource code uses a per-pool lock to protect its internal state. If any pool functions are called in an interrupt context, the caller must block all interrupts that might cause the code to be reentered. Additionally, the functions **pool_init**() and **pool_destroy**() should never be called in interrupt context.

DIAGNOSTICS

Pool usage logs can be enabled by defining the compile-time option `POOL_DIAGNOSTIC`.

CODE REFERENCES

The pool manager is implemented in the file `sys/kern/subr_pool.c`.

SEE ALSO

`free`(9), `malloc`(9), `memoryallocators`(9), `pool_cache`(9), `uvm`(9)

HISTORY

The NetBSD pool manager appeared in NetBSD 1.4.

NAME

pool_cache, **pool_cache_init**, **pool_cache_destroy**, **pool_cache_get_paddr**, **pool_cache_get**, **pool_cache_put_paddr**, **pool_cache_put**, **pool_cache_destruct_object**, **pool_cache_invalidate**, **pool_cache_sethiwat**, **pool_cache_setlowat** — resource-pool cache manager

SYNOPSIS

```
#include <sys/pool.h>

pool_cache_t
pool_cache_init(size_t size, u_int align, u_int align_offset, int flags,
    const char *name, struct pool_allocator *palloc, int ipl,
    int (*ctor)(void *, void *, int), void (*dtor)(void *, void *),
    void *arg);

void
pool_cache_destroy(pool_cache_t pc);

void *
pool_cache_get_paddr(pool_cache_t pc, int flags, paddr_t *pap);

void *
pool_cache_get(pool_cache_t pc, int flags);

void
pool_cache_put_paddr(pool_cache_t pc, void *object, paddr_t pa);

void
pool_cache_put(pool_cache_t pc, void *object);

void
pool_cache_destruct_object(pool_cache_t pc, void *object);

void
pool_cache_invalidate(pool_cache_t pc);

void
pool_cache_sethiwat(pool_cache_t pc, int nitems);

void
pool_cache_setlowat(pool_cache_t pc, int nitems);
```

DESCRIPTION

These utility routines provide management of pools of fixed-sized areas of memory. Resource pools set aside an amount of memory for exclusive use by the resource pool owner. This can be used by applications to guarantee the availability of a minimum amount of memory needed to continue operation independent of the memory resources currently available from the system-wide memory allocator.

Global and per-CPU caches of constructed objects are maintained. The two levels of cache work together to allow for low overhead allocation and release of objects, and improved L1/L2/L3 hardware cache locality in multiprocessor systems.

FUNCTIONS

pool_cache_init(*pc*, *pp*, *ctor*, *dtor*, *arg*)

Allocate and initialize a pool cache. The arguments are:

size

Specifies the size of the memory items managed by the pool.

align

Specifies the memory address alignment of the items returned by **pool_cache_get**(). This argument must be a power of two. If zero, the alignment defaults to an architecture-specific natural alignment.

align_offset

The offset within an item to which the *align* parameter applies.

flags

Should be set to zero or PR_NOTOUCH. If PR_NOTOUCH is given, free items are never used to keep internal state so that the pool can be used for non memory backed objects.

name

The name used to identify the object in diagnostic output.

palloc

Should be typically be set to NULL, instructing **pool_cache_init**() to select an appropriate back-end allocator. Alternate allocators can be used to partition space from arbitrary sources. Use of alternate allocators is not documented here as it is not a stable, endorsed part of the API.

ipl

Specifies an interrupt priority level that will block all interrupt handlers that could potentially access the pool. The **pool_cache** facility provides its own synchronization. The users of any given **pool_cache** need not provide additional synchronization for access to it.

ctor

Specifies a constructor used to initialize newly allocated objects. If no constructor is required, specify NULL.

dtor

Specifies a destructor used to destroy cached objects prior to their release to backing store. If no destructor is required, specify NULL.

arg

This value of this argument will be passed to both the constructor and destructor routines.

pool_cache_destroy(*pc*)

Destroy a pool cache. All other access to the cache must be stopped before this call can be made. *pc*.

pool_cache_get_paddr(*pc*, *flags*, *pap*)

Get an object from a pool cache *pc*. If *pap* is not NULL, physical address of the object or POOL_PADDR_INVALID will be returned via it. *flags* will be passed to **pool_get**() function of the backing pool(9) and the object constructor specified when the pool cache is created by **pool_cache_init**().

pool_cache_get(*pc*, *flags*)

pool_cache_get() is the same as **pool_cache_get_paddr**() with NULL *pap* argument. It's implemented as a macro.

pool_cache_put_paddr(*pc*, *object*, *pa*)

Put an object *object* back to the pool cache *pc*. *pa* should be physical address of the object *object* or POOL_PADDR_INVALID. *pp*. If the number of available items in the backing pool exceeds the maximum pool size set by **pool_cache_sethiwat**() and there are no outstanding requests for pool items, the excess items will be returned to the system.

pool_cache_put(*pc*, *object*)

pool_cache_put() is the same as **pool_cache_put_paddr**() with POOL_PADDR_INVALID *pa* argument. It's implemented as a macro.

pool_cache_destruct_object(*pc*, *object*)

Force destruction of an object *object* and its release back into the pool.

pool_cache_invalidate(*pc*)

Invalidate a pool cache *pc*. Destruct and release all objects in the global cache. Per-CPU caches will not be invalidated by this call, meaning that it is still possible to allocate "stale" items from the cache. If relevant, the user must check for this condition when allocating items.

pool_cache_sethiwat(*pc*, *nitems*)

A pool will attempt to increase its resource usage to keep up with the demand for its items. Conversely, it will return unused memory to the system should the number of accumulated unused items in the pool exceed a programmable limit. The limits for the minimum and maximum number of items which a pool should keep at hand are known as the high and low **watermarks**.

The function **pool_cache_sethiwat**() sets the backing pool's high water mark. As items are returned and the total number of pages in the pool is larger than the maximum set by this function, any completely unused pages are released immediately. If this function is not used to specify a maximum number of items, the pages will remain associated with the pool until the system runs low on memory, at which point the VM system will try to reclaim unused pages.

pool_cache_setlowat(*pc*, *nitems*)

Set the minimum number of items to keep in the pool. The number pages in the pool will not decrease below the required value to accommodate the minimum number of items specified by this function.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing the **pool_cache** subsystem can be found. All pathnames are relative to /usr/src.

The **pool_cache** subsystem is implemented within the file sys/kern/subr_pool.c.

SEE ALSO

intro(9), kmem_alloc(9), kmem_free(9), memoryallocators(9), pool(9)

NAME

powerhook_establish, **powerhook_disestablish** — add or remove a power change hook

SYNOPSIS

void *
powerhook_establish(*const char *name*, *void (*fn)(int why, void *a)*,
*void *arg*);

void
powerhook_disestablish(*void *cookie*);

DESCRIPTION

The **powerhook_establish**() function adds *fn* of the list of hooks invoked by dopowerhooks(9) at power change. When invoked, the hook function *fn* will be passed the new power state as the first argument and *arg* as its second argument.

The **powerhook_disestablish**() function removes the hook described by the opaque pointer *cookie* from the list of hooks to be invoked at power change. If *cookie* is invalid, the result of **powerhook_disestablish**() is undefined.

Power hooks should be used to perform activities that must happen when the power situation to the computer changes. Because of the environment in which they are run, power hooks cannot rely on many system services (including file systems, and timeouts and other interrupt-driven services). The power hooks are typically executed from an interrupt context.

The different reasons for calling the power hooks are: suspend, standby, and resume. The reason is reflected in the *why* argument and the values PWR_SOFTSUSPEND, PWR_SUSPEND, PWR_SOFTSTANDBY, PWR_STANDBY, PWR_SOFTRESUME, and PWR_RESUME. It calls with PWR_SOFTxxx in the normal priority level while the other callings are protected with splhigh(9). At suspend the system is going to lose (almost) all power, standby retains some power (e.g., minimal power to USB devices), and at resume power is back to normal.

RETURN VALUES

If successful, **powerhook_establish**() returns an opaque pointer describing the newly-established power hook. Otherwise, it returns NULL.

SEE ALSO

dopowerhooks(9)

NAME

ppi — user-space interface to ppbus parallel port

SYNOPSIS

```
#include <sys/ioctl.h>
#include <dev/ppbus/ppi.h>
#include <dev/ppbus/ppbus_conf.h>
```

DESCRIPTION

All I/O on the **ppi** interface is performed using **ioctl**() calls. Each command takes a single *uint8_t* argument, transferring one byte of data. The following commands are available:

PPIGDATA, PPISDATA
: Get and set the contents of the data register.

PPIGSTATUS, PPISSTATUS
: Get and set the contents of the status register.

PPIGCTRL, PPISCTRL
: Get and set the contents of the control register. The following defines correspond to bits in this register. Setting a bit in the control register drives the corresponding output low.
  STROBE
  AUTOFEED
  nINIT
  SELECTIN
  PCD

PPIGEPP, PPISEPP
: Get and set the contents of the EPP control register.

PPIGECR, PPISECR
: Get and set the contents of the ECP control register.

PPIGFIFO, PPISFIFO
: Read and write the ECP FIFO (8-bit operations only).

EXAMPLES

To present the value 0x5a to the data port, drive STROBE low and then high again, the following code fragment can be used:

```
int            fd;
uint8_t val;

val = 0x5a;
ioctl(fd, PPISDATA, &val);
ioctl(fd, PPIGCTRL, &val);
val |= STROBE;
ioctl(fd, PPISCTRL, &val);
val &= ~STROBE;
ioctl(fd, PPISCTRL, &val);
```

SEE ALSO

ioctl(2), atppc(4), io(4), ppbus(4), ppi(4)

HISTORY

ppi originally appeared in FreeBSD.

AUTHORS

This manual page is based on the FreeBSD **ppi** manual page and was updated for the NetBSD port by Gary Thorpe.

BUGS

The inverse sense of signals is confusing.

The **ioctl**() interface is slow, and there is no way (yet) to chain multiple operations together.

The headers required for user applications are not installed as part of the standard system.

NAME

ppsratecheck — function to help implement rate-limited actions

SYNOPSIS

```
#include <sys/time.h>

int
ppsratecheck(struct timeval *lasttime, int *curpps, int maxpps);
```

DESCRIPTION

The **ppsratecheck**() function provides easy way to perform packet-per-sec, or event-per-sec, rate limitation. The motivation for implementing **ppsratecheck**() was to provide a mechanism that could be used to add rate limitation to network packet output. For certain network packets, we may want to impose rate limitation, to avoid denial-of-service attack possibilities.

maxpps specifies maximum permitted packets, or events, per second. If **ppsratecheck**() is called more than *maxpps* times in a given one second period, the function will return 0, indicating that we exceeded the limit. If we are below the limit, the function will return 1. If *maxpps* is set to 0, the function will always return 0 (no packets/events are permitted). Negative *maxpps* indicates that rate limitation is disabled, and *ppsratecheck* will always return 1.

curpps and *lasttime* are used to maintain the number of recent calls. *curpps* will be incremented every time **ppsratecheck**() is called, and will be reset whenever necessary.

SEE ALSO

log(9), printf(9), ratecheck(9), time_second(9)

HISTORY

The **ppsratecheck**() function appeared in NetBSD 1.5.

NAME

preempt, **yield** — general preempt and yield functions

SYNOPSIS

```
#include <sys/sched.h>

void
preempt(void);

#include <sys/proc.h>

void
yield(void);
```

DESCRIPTION

The **preempt**() function puts the current LWP back on the system run queue and performs an involuntary context switch. The **yield**() function is mostly same as **preempt**(), except that it performs a voluntary context switch.

These functions drop the kernel lock before switching and re-acquire it before returning.

NAME

prop_array_copyin_ioctl, **prop_array_copyout_ioctl**, **prop_dictionary_copyin_ioctl**, **prop_dictionary_copyout_ioctl** — Copy property lists to and from kernel space

SYNOPSIS

```
#include <prop/proplib.h>

int
prop_array_copyin_ioctl(const struct plistref *pref, const u_long cmd,
    prop_array_t *arrayp);

int
prop_array_copyout_ioctl(struct plistref *pref, const u_long cmd,
    prop_array_t array);

int
prop_dictionary_copyin_ioctl(const struct plistref *pref,
    const u_long cmd, prop_dictionary_t *dictp);

int
prop_dictionary_copyout_ioctl(struct plistref *pref, const u_long cmd,
    prop_dictionary_t dict);
```

DESCRIPTION

The **prop_array_copyin_ioctl**, **prop_array_copyout_ioctl**, **prop_dictionary_copyin_ioctl**, and **prop_dictionary_copyout_ioctl** functions implement the kernel side of a protocol for sending property lists to and from the kernel using ioctl(2).

A kernel ioctl routine receiving or returning a property list will be passed a pointer to a *struct plistref*. This structure encapsulates the reference to the property list in externalized form.

RETURN VALUES

If successful, functions return zero. Otherwise, an error number will be returned to indicate the error.

ERRORS

prop_array_copyin_ioctl() and **prop_dictionary_copyin_ioctl**() will fail if:

[EFAULT]	Bad address
[EIO]	Input/output error
[ENOMEM]	Cannot allocate memory
[ENOTSUP]	Not supported

prop_array_copyout_ioctl() and **prop_dictionary_copyout_ioctl**() will fail if:

[EFAULT]	Bad address
[ENOMEM]	Cannot allocate memory
[ENOTSUP]	Not supported

EXAMPLES

The following (simplified) example demonstrates using **prop_dictionary_copyin_ioctl**() and **prop_dictionary_copyout_ioctl**() in an ioctl routine:

```
extern prop_dictionary_t fooprops;

int
fooioctl(dev_t dev, u_long cmd, caddr_t data, int flag, struct lwp *l)
{
    prop_dictionary_t dict, odict;
    int error;

    switch (cmd) {
    case FOOSETPROPS: {
        const struct plistref *pref = (const struct plistref *) data;
        error = prop_dictionary_copyin_ioctl(pref, cmd, &dict);
        if (error)
                return (error);
        odict = fooprops;
        fooprops = dict;
        prop_object_release(odict);
        break;
      }

    case FOOGETPROPS: {
        struct plistref *pref = (struct plistref *) data;
        error = prop_dictionary_copyout_ioctl(pref, cmd, fooprops);
        break;
      }

    default:
        return (EPASSTHROUGH);
    }
    return (error);
}
```

SEE ALSO

`prop_array`(3), `prop_dictionary`(3), `prop_send_ioctl`(3), `proplib`(3)

HISTORY

The **`proplib`** property container object library first appeared in NetBSD 4.0.

NAME

putter — Pass-to-Userspace Transporter

DESCRIPTION

The **putter** subsystem is used for request-response handling of userspace components. It currently provides routines for associating a file descriptor with a subsystem data structure instance and I/O routines. Users of the facility must fill out the callbacks in *struct putter_ops* to integrate with **putter**.

SEE ALSO

pud(4), puffs(4)

BUGS

Under construction. Interfaces may and will change.

NAME

radio — interface between low and high level radio drivers

DESCRIPTION

The radio device driver is divided into a high level, hardware independent layer, and a low level hardware dependent layer. The interface between these is the *radio_hw_if* structure.

```
struct radio_hw_if {
        int     (*open)(void *, int, int, struct lwp *);
        int     (*close)(void *, int, int, struct lwp *);
        int     (*get_info)(void *, struct radio_info *);
        int     (*set_info)(void *, struct radio_info *);
        int     (*search)(void *, int);
};
```

The high level radio driver attaches to the low level driver when the latter calls *radio_attach_mi*. This call should be

```
    void
    radio_attach_mi(rhwp, hdlp, dev)
        struct radio_hw_if *rhwp;
        void *hdlp;
        struct device *dev;
```

The *radio_hw_if* struct is as shown above. The *hdlp* argument is a handle to some low level data structure. It is sent as the first argument to all the functions in *radio_hw_if* when the high level driver calls them. *dev* is the device struct for the hardware device.

The fields of *radio_hw_if* are described in some more detail below.

```
int open (void *, int flags, int fmt, struct lwp *p);
  Optional.
  Is called when the radio device is opened.
  Returns 0 on success, otherwise an error code.

int close (void *, int flags, int fmt, struct lwp *p);
  Optional.
  Is called when the radio device is closed.
  Returns 0 on success, otherwise an error code.

int get_info (void *, struct radio_info *);
  Fill the radio_info struct.
  Returns 0 on success, otherwise an error code.

int set_info (void *, struct radio_info *);
  Set values from the radio_info struct.
  Returns 0 on success, otherwise an error code.

int search (void *, int);
  Returns 0 on success, otherwise an error code.
```

SEE ALSO

radio(4)

NAME

ras_lookup, **ras_fork**, **ras_purgeall** — restartable atomic sequences

SYNOPSIS

```
#include <sys/types.h>
#include <sys/proc.h>
#include <sys/ras.h>

void *
ras_lookup(struct proc *p, void *addr);

int
ras_fork(struct proc *p1, struct proc *p2);

int
ras_purgeall(struct proc *p);
```

DESCRIPTION

Restartable atomic sequences are user code sequences which are guaranteed to execute without preemption. This property is assured by checking the set of restartable atomic sequences registered for a process during `cpu_switch`(9). If a process is found to have been preempted during a restartable sequence, then its execution is rolled-back to the start of the sequence by resetting its program counter saved in its process control block (PCB).

The RAS functionality is provided by a combination of the machine-independent routines discussed in this page and a machine-dependent component in `cpu_switch`(9). A port which supports restartable atomic sequences will define __HAVE_RAS in `machine/types.h` for machine-independent code to conditionally provide RAS support.

A complicated side-effect of restartable atomic sequences is their interaction with the machine-dependent `ptrace`(2) support. Specifically, single-step traps and/or the emulation of single-stepping must carefully consider the effect on restartable atomic sequences. A general solution is to ignore these traps or disable them within restartable atomic sequences.

FUNCTIONS

The functions which operate on restartable atomic sequences are:

ras_lookup(*p*, *addr*)
: This function searches the registered restartable atomic sequences for process *p* which contain the user address *addr*. If the address *addr* is found within a RAS, then the restart address of the RAS is returned, otherwise −1 is returned.

ras_fork(*p1*, *p2*)
: This function is used to copy all registered restartable atomic sequences for process *p1* to process *p2*. It is primarily called from `fork1`(9) when the sequences are inherited from the parent by the child.

ras_purgeall(*p*)
: This function is used to remove all registered restartable atomic sequences for process *p*. It is primarily used to remove all registered restartable atomic sequences for a process during `exec`(3) and by `rasctl`(2).

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the RAS functionality can be found. All pathnames are relative to `/usr/src`.

The RAS framework itself is implemented within the file `sys/kern/kern_ras.c`. Data structures and function prototypes for the framework are located in `sys/sys/ras.h`. Machine-dependent portions are implemented within `cpu_switch`(9) in the machine-dependent file `sys/arch/<arch>/<arch>/locore.S`.

SEE ALSO

`rasctl`(2), `cpu_switch`(9), `fork1`(9)

HISTORY

The RAS functionality first appeared in NetBSD 2.0.

NAME

rasops, **rasops_init**, **rasops_reconfig** — raster display operations

SYNOPSIS

```
#include <dev/wscons/wsdisplayvar.h>
#include <dev/rasops/rasops.h>

int
rasops_init(struct rasops_info *ri, int wantrows, int wantcols);

int
rasops_reconfig(struct rasops_info *ri, int wantrows, int wantcols);
```

DESCRIPTION

The **rasops** subsystem is a set of raster operations for wscons(9).

The primary data type for using the raster operations is the *rasops_info* structure in dev/rasops/rasops.h:

```
struct rasops_info {

        /*
         * These must be filled in by the caller
         */
        int     ri_depth;       /* depth in bits */
        u_char  *ri_bits;       /* ptr to bits */
        int     ri_width;       /* width (pels) */
        int     ri_height;      /* height (pels) */
        int     ri_stride;      /* stride in bytes */

        /*
         * If you want shadow framebuffer support, point ri_hwbits
         * to the real framebuffer, and ri_bits to the shadow framebuffer
         */
        u_char  *ri_hwbits;

        /*
         * These can optionally be left zeroed out. If you fill ri_font,
         * but aren't using wsfont, set ri_wsfcookie to -1.
         */
        struct  wsdisplay_font *ri_font;
        int     ri_wsfcookie;  /* wsfont cookie */
        void    *ri_hw;        /* driver private data */
        int     ri_crow;       /* cursor row */
        int     ri_ccol;       /* cursor column */
        int     ri_flg;        /* various operational flags */

        /*
         * These are optional and will default if zero. Meaningless
         * on depths other than 15, 16, 24 and 32 bits per pel. On
         * 24 bit displays, ri_{r,g,b}num must be 8.
         */
        u_char  ri_rnum;       /* number of bits for red */
        u_char  ri_gnum;       /* number of bits for green */
```

```
        u_char  ri_bnum;       /* number of bits for blue */
        u_char  ri_rpos;       /* which bit red starts at */
        u_char  ri_gpos;       /* which bit green starts at */
        u_char  ri_bpos;       /* which bit blue starts at */

        /*
         * These are filled in by rasops_init()
         */
        int     ri_emuwidth;   /* width we actually care about */
        int     ri_emuheight;  /* height we actually care about */
        int     ri_emustride;  /* bytes per row we actually care about */
        int     ri_rows;       /* number of rows (characters) */
        int     ri_cols;       /* number of columns (characters) */
        int     ri_delta;      /* row delta in bytes */
        int     ri_pelbytes;   /* bytes per pel (may be zero) */
        int     ri_fontscale;  /* fontheight * fontstride */
        int     ri_xscale;     /* fontwidth * pelbytes */
        int     ri_yscale;     /* fontheight * stride */
        u_char  *ri_origbits;  /* where screen bits actually start */
        int     ri_xorigin;    /* where ri_bits begins (x) */
        int     ri_yorigin;    /* where ri_bits begins (y) */
        int32_t ri_devcmap[16]; /* color -> framebuffer data */

        /*
         * The emulops you need to use, and the screen caps for wscons
         */
        struct  wsdisplay_emulops ri_ops;
        int     ri_caps;

        /*
         * Callbacks so we can share some code
         */
        void    (*ri_do_cursor)(struct rasops_info *);
};
```

Valid values for the *ri_flg* member are:

RI_FULLCLEAR	eraserows() hack to clear full screen
RI_FORCEMONO	monochrome output even if we can do color
RI_BSWAP	framebuffer endianness doesn't match CPU
RI_CURSOR	cursor is switched on
RI_CLEAR	clear display on startup
RI_CENTER	center onscreen output
RI_CURSORCLIP	cursor is currently clipped
RI_CFGDONE	**rasops_reconfig**() completed successfully

FUNCTIONS

rasops_init(*ri*, *wantrows*, *wantcols*)

Initialise a *rasops_info* descriptor. The arguments *wantrows* and *wantcols* are the number of rows and columns we'd like. In terms of optimization, fonts that are a multiple of 8 pixels wide work the best.

rasops_reconfig(*ri*, *wantrows*, *wantcols*)
Reconfigure a *rasops_info* descriptor because parameters have changed in some way. The arguments *wantrows* and *wantcols* are the number of rows and columns we'd like. If calling **rasops_reconfig**() to change the font and ri_wsfcookie ≥ 0, you must call **wsfont_unlock**() on it, and reset it to -1 (or a new, valid cookie).

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the rasops subsystem can be found. All pathnames are relative to `/usr/src`.

The rasops subsystem is implemented within the directory `sys/dev/rasops`. The **rasops** module itself is implemented within the file `sys/dev/rasops/rasops.c`.

SEE ALSO

`intro`(9), `wscons`(9), `wsdisplay`(9), `wsfont`(9)

HISTORY

The **rasops** subsystem appeared in NetBSD 1.5.

AUTHORS

The **rasops** subsystem was written by Andrew Doran ⟨ad@NetBSD.org⟩.

NAME

ratecheck — function to help implement rate-limited actions

SYNOPSIS

```
#include <sys/time.h>

int
ratecheck(struct timeval *lasttime, const struct timeval *mininterval);
```

DESCRIPTION

The **ratecheck**() function provides a simple time interval check which can be used when implementing time-based rate-limited actions. If the difference between the current monotonically-increasing system time (*mono_time*) and *lasttime* is less than the value given by the *mininterval* argument, zero is returned. Otherwise, *lasttime* is set to the current time and a non-zero value is returned.

The motivation for implementing **ratecheck**() was to provide a mechanism that could be used to add rate limiting to diagnostic message output. If printed too often, diagnostic messages can keep the system from doing useful work. If the repeated messages can be caused by deliberate user action or network events, they can be exploited to cause denial of system service.

Note that using a very short time interval (less than a second) for *mininterval* defeats the purpose of this function. (It doesn't take much to flood a 9600 baud serial console with output, for instance.)

EXAMPLES

Here is a simple example of use of the **ratecheck**() function:

```
/*
 * The following variables could be global, in a device softc, etc.,
 * depending on the exact usage.
 */
struct timeval drv_lasterr1time;   /* time of last err1 message */
long drv_err1count;                /* # of err1 errs since last msg */
struct timeval drv_lasterr2time;   /* time of last err2 message */
long drv_err2count;                /* # of err2 errs since last msg */

/*
 * The following variable will often be global or shared by all
 * instances of a driver.  It should be initialized, so it can be
 * patched.  Allowing it to be set via an option might be nice,
 * but could lead to an insane proliferation of options.
 */
struct timeval drv_errintvl = { 5, 0 };         /* 5 seconds */

/* error handling/reporting function */
void
drv_errhandler(int err1, int err2)
{

        /*
         * Note that you should NOT use the same last-event
         * time variable for dissimilar messages!
         */
        if (err1) {
                /* handle err1 condition */
```

```
                ...

                drv_err1count++;
                if (ratecheck(&drv_lasterr1notice,
                    &drv_errinterval)) {
                        printf("drv: %ld err1 errors occurred",
                            drv_err1count);
                        drv_err1count = 0;
                }
        }
        if (err2) {
                /* handle err2 condition */
                ...

                drv_err2count++;
                if (ratecheck(&drv_lasterr2notice,
                    &drv_errinterval)) {
                        printf("drv: %ld err2 errors occurred",
                            drv_err2count);
                        drv_err2count = 0;
                }
        }
}
```

SEE ALSO

log(9), ppsratecheck(9), printf(9), time_second(9)

HISTORY

The **ratecheck**() function appeared in NetBSD 1.5.

BUGS

ratecheck() may not work as expected, if *mininterval* is less than the hardware clock interrupt interval (1/hz).

NAME

resettodr — set battery-backed clock from system time

SYNOPSIS

void
resettodr(*void*);

DESCRIPTION

The **resettodr**() function sets the system's battery-backed clock based on the current system time.

SEE ALSO

clock_secs_to_ymdhms(9), inittodr(9), time_second(9)

NAME

RND, **rnd_attach_source**, **rnd_detach_source**, **rnd_add_data**, **rnd_add_uint32** — functions to make a device available for entropy collection

SYNOPSIS

```
#include <sys/rnd.h>

void
rnd_attach_source(rndsource_element_t *rnd_source, char *devname,
    uint32_t source_type, uint32_t flags);

void
rnd_detach_source(rndsource_element_t *rnd_source);

void
rnd_add_data(rndsource_element_t *rnd_source, void *data, uint32_t len,
    uint32_t entropy);

void
rnd_add_uint32(rndsource_element_t *rnd_source, uint32_t datum);
```

DESCRIPTION

These **RND** functions make a device available for entropy collection for /dev/random.

Ideally the first argument *rnd_source* of these functions gets included in the devices' entity struct, but any means to permanently (static) attach one such argument to one incarnation of the device is ok. Do not share *rnd_source* structures between two devices.

rnd_attach_source(*rndsource_element_t *rnd_source*, *char *devname*, *uint32_t source_type*, *uint32_t flags*)
: This function announces the availability of a device for entropy collection. It must be called before the source struct pointed to by *rnd_source* is used in any of the following functions.

  devname is the name of the device. It is used to print a message (if the kernel is compiled with "options RND_VERBOSE") and also for status information printed with rndctl(8).

  source_type is RND_TYPE_NET for network devices, RND_TYPE_DISK for physical disks, RND_TYPE_TAPE for a tape drive, and RND_TYPE_TTY for a tty. RND_TYPE_UNKNOWN is not to be used as a type. It is used internally to the rnd system.

  flags are the logical OR of RND_FLAG_NO_COLLECT (don't collect or estimate) RND_FLAG_NO_ESTIMATE (don't estimate) to control the default setting for collection and estimation. Note that devices of type RND_TYPE_NET default to RND_FLAG_NO_ESTIMATE.

rnd_detach_source(*rndsource_element_t *rnd_source*)
: This function disconnects the device from entropy collection.

rnd_add_uint32(*rndsource_element_t *rnd_source*, *uint32_t datum*)
: This function adds the value of *datum* to the entropy pool. No entropy is assumed to be collected from this value, it merely helps stir the entropy pool. All entropy is gathered from jitter between the timing of events.

  Note that using a constant for *datum* does not weaken security, but it does not help. Try to use something that can change, such as an interrupt status register which might have a bit set for receive ready or transmit ready, or other device status information.

  To allow the system to gather the timing information accurately, this call should be placed within the actual hardware interrupt service routine. Care must be taken to ensure that the interrupt was

actually serviced by the interrupt handler, since on some systems interrupts can be shared.

This function loses nearly all usefulness if it is called from a scheduled software interrupt. If that is the only way to add the device as an entropy source, don't.

If it is desired to mix in the *datum* and to add in a timestamp, but not to actually estimate entropy from a source of randomness, passing `NULL` for *rnd_source* is permitted, and the device does not need to be attached.

rnd_add_data(*rndsource_element_t *rnd_source*, *void *data*, *uint32_t len*, *uint32_t entropy*)
adds (hopefully) random *data* to the entropy pool. *len* is the number of bytes in *data* and *entropy* is an "entropy quality" measurement. If every bit of *data* is known to be random, *entropy* is the number of bits in *data*.

Timing information is also used to add entropy into the system, using inter-event timings.

If it is desired to mix in the *data* and to add in a timestamp, but not to actually estimate entropy from a source of randomness, passing `NULL` for *rnd_source* is permitted, and the device does not need to be attached.

FILES

These functions are declared in src/sys/sys/rnd.h and defined in src/sys/dev/rnd.c.

SEE ALSO

`rnd`(4), `rndctl`(8)

HISTORY

The random device was introduced in NetBSD 1.3.

AUTHORS

This implementation was written by Michael Graff <explorer@flame.org> using ideas and algorithms gathered from many sources, including the driver written by Ted Ts'o.

BUGS

The only good sources of randomness are quantum mechanical, and most computers avidly avoid having true sources of randomness included. Don't expect to surpass "pretty good".

NAME

rssadapt, **ieee80211_rssadapt_choose**, **ieee80211_rssadapt_input**, **ieee80211_rssadapt_lower_rate**, **ieee80211_rssadapt_raise_rate**, **ieee80211_rssadapt_updatestats** — rate adaptation based on received signal strength

SYNOPSIS

```
#include <net80211/ieee80211_var.h>
#include <net80211/ieee80211_rssadapt.h>

void
ieee80211_rssadapt_input(struct ieee80211com *ic,
    struct ieee80211_node *ni, struct ieee80211_rssadapt *ra, int rssi);

void
ieee80211_rssadapt_lower_rate(struct ieee80211com *ic,
    struct ieee80211_node *ni, struct ieee80211_rssadapt *ra,
    struct ieee80211_rssdesc *id);

void
ieee80211_rssadapt_raise_rate(struct ieee80211com *ic,
    struct ieee80211_rssadapt *ra, struct ieee80211_rssdesc *id);

void
ieee80211_rssadapt_updatestats(struct ieee80211_rssadapt *ra);

int
ieee80211_rssadapt_choose(struct ieee80211_rssadapt *ra,
    struct ieee80211_rateset *rs, struct ieee80211_frame *wh, u_int len,
    int fixed_rate, const char *dvname, int do_not_adapt);
```

DESCRIPTION

The **rssadapt** module provides rapid adaptation of transmission data rate to 802.11 device drivers based on received-signal strength (RSS). A driver needs only to provide **rssadapt** with indications of RSS and failure/success of transmissions for each 802.11 client or peer. For each transmit packet, **rssadapt** chooses the transmission data rate that offers the best expected throughput, given the packet's length and destination.

rssadapt models an 802.11 channel very simply (see also the **BUGS** section). It assumes that the packet-error rate (PER) is determined by the signal-to-noise ratio (S/N) at the receiver, the transmission data rate, and the packet length. The S/N determines the choice of data rate that yields the lowest PER for all packets of a certain length.

FUNCTIONS

ieee80211_rssadapt_choose(*ra*, *rs*, *wh*, *len*, *fixed_rate*, *dvname*, *do_not_adapt*)

Choose the transmission data rate for a packet.

ra Ordinarily, the **rssadapt** state object belonging to the node which is the packet destination. However, if the destination is a broadcast/multicast address, then *ra* belongs to the BSS node, *ic->ic_bss*.

rs A list of eligible data rates for the node; for example, the rates negotiated when the node associated with the network.

len The packet length in bytes, including the 802.11 header and frame check sequence (FCS).

fixed_rate If the operator has set the data rate using, for example, **ifconfig wi0 media ds1**, then *fixed_rate* tells the index of that rate in *rs*. **rssadapt** obeys a fixed data rate whenever the 802.11 standard allows it: sometimes the standard requires multicast/broadcast packets to be transmitted at a so-called "basic rate".

dvname The device driver uses *dvname* to indicate the name of the interface for the purpose of diagnostic and debug messages. The driver sets *dvname* to NULL when no messages are desired.

do_not_adapt If *do_not_adapt* is non-zero, then **ieee80211_rssadapt_choose**() will choose the highest rate in *rs* that suits the destination, regardless of the RSS.

The return value of **ieee80211_rssadapt_choose**() is an index into *rs*, indicating its choice of transmit data rate.

ieee80211_rssadapt_input(*ic*, *ni*, *ra*, *rssi*)

The RSS serves as a rough estimate of the S/N at each node. A driver provides RSS updates using **ieee80211_rssadapt_input**(), whose arguments are:

ic The wireless interface's 802.11 state object.

ni The 802.11 node whose RSS the driver is updating.

ra The node's **rssadapt** state object.

rssi The node's received signal strength indication. The range of *rssi* is from 0 to 255.

ieee80211_rssadapt_lower_rate(*ic*, *ni*, *ra*, *id*)

ieee80211_rssadapt_raise_rate(*ic*, *ra*, *id*)

Drivers call **ieee80211_rssadapt_raise_rate**() and **ieee80211_rssadapt_lower_rate**() to indicate transmit successes and failures, respectively.

ic The 802.11 state object.

ni The neighbor to whom the driver transmitted.

ra The neighbor's **rssadapt** state object.

id DIsplays statistics on the transmission attempt.

ieee80211_rssadapt_updatestats(*ra*)

An 802.11 node is eligible for its RSS thresholds to decay every 1/10 to 10 seconds. It is eligible more often (every 1/10 second) at high packet rates, and less often (every 10 seconds) at low packet rates. A driver assists **rssadapt** in tracking the exponential-average packet rate by calling **ieee80211_rssadapt_updatestats**() every 1/10th second for each node's *ieee80211_rssadapt* object.

ra The neighbor's **rssadapt** state object.

ALGORITHM

rssadapt monitors the RSS from neighboring 802.11 nodes, recording the exponential average RSS in each neighbor's *ieee80211_rssadapt* structure. **rssadapt** uses transmit success/failure feedback from the device driver to fill a table of RSS thresholds. The table is indexed by packet size, *L*, and a data rate, *R*, to find out the minimum exponential-average RSS that a node must show before **rssadapt** will indicate that a packet *L* bytes long can be transmitted R bits per second with optimal expected throughput. When the driver indicates a unicast packet is transmitted unsuccessfully (that is, the NIC received no ACK for the packet), **rssadapt** will move the corresponding RSS threshold toward the exponential average RSSI at the time of transmission. Thus several consecutive transmit failures for the same $\langle L, R \rangle$ tuple will ensure that the RSS threshold rises high enough that rate *R* is abandoned for packets *L* bytes long. When the driver indicates a successful transmission, the RSS threshold corresponding to the same packet length, but the next higher data rate, is lowered slightly. The RSS threshold is said to "decay". This ensures that occasionally **rssadapt** indicates the driver should try the next higher data rate, just in case conditions at the

receiver have changed (for example, noise levels have fallen) and a higher data rate can be supported at the same RSS level.

The rate of decay is controlled. In an interval of 1/10th second to 10 seconds, only one RSS threshold per neighbor may decay. The interval is connected to the exponential-average rate that packets are being transmitted. At high packet rates, the interval is shortest. It is longest at low packet rates. The rationale for this is that RSS thresholds should not decay rapidly if there is no information from packet transmissions to counteract their decay.

DATA STRUCTURES

An *ieee80211_rssdesc* describes a transmission attempt.

```
struct ieee80211_rssdesc {
        u_int                    id_len;
        u_int                    id_rateidx;
        struct ieee80211_node   *id_node;
        u_int8_t                 id_rssi;
};
```

id_len is the length, in bytes, of the transmitted packet. *id_node* points to the neighbor's *ieee8021_node*, and *id_rssi* is the exponential-average RSS at the time the packet was transmitted. *id_rateidx* is an index into the destination-neighbor's rate-set, *id_node->ni_rates*, indicating the transmit data rate for the packet.

An *ieee80211_rssadapt* contains the rate-adaptation state for a neighboring 802.11 node. Ordinarily a driver will "subclass" *ieee80211_node*. The *ieee80211_rssadapt* structure will be a subclass member. In this way, every node's **rssadapt** condition is independently tracked and stored in its node object.

```
struct ieee80211_rssadapt {
        u_int16_t               ra_avg_rssi;
        u_int32_t               ra_nfail;
        u_int32_t               ra_nok;
        u_int32_t               ra_pktrate;
        u_int16_t               ra_rate_thresh[IEEE80211_RSSADAPT_BKTS]
                                              [IEEE80211_RATE_SIZE];
        struct timeval          ra_last_raise;
        struct timeval          ra_raise_interval;
};
```

ra_avg_rssi is the exponential-average RSS, shifted left 8 bits. *ra_nfail* tells the number of transmit failures in the current update interval. *ra_nok* tells the number of transmit successes in the current update interval. *ra_pktrate* tells the exponential average number of transmit failure/success indications over past update intervals. This approximates the rate of packet-transmission. *ra_rate_thresh* contains RSS thresholds that are indexed by ⟨packet length, data rate⟩ tuples. When this node's exponential-average RSS exceeds *ra_rate_thresh[i][j]*, then packets at most 128 x 8^i bytes long are eligible to be transmitted at the rate indexed by j. *ra_last_raise* and *ra_raise_interval* are used to control the rate that RSS thresholds "decay". *ra_last_raise* indicates when **ieee80211_rssadapt_raise_rate**() was last called. *ra_raise_interval* tells the minimum period between consecutive calls to **ieee80211_rssadapt_raise_rate**(). If **ieee80211_rssadapt_raise_rate**() is called more than once in any period, the second and subsequent calls are ignored.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using **rssadapt** can be found. All pathnames are relative to `/usr/src`.

The code for **rssadapt** is in the file `sys/net80211/ieee80211_rssadapt.c`.

`wi`(4) contains a reference implementation. See `sys/dev/ic/wi.c`.

SEE ALSO

`wi`(4)

Javier del Prado Pavon and Sunghyun Choi, "Link Adaptation Strategy for IEEE 802.11 WLAN via Received Signal Strength Measurement", *ICC'03*, pp. 1108-1113, May 2003.

HISTORY

rssadapt first appeared in NetBSD 3.0.

AUTHORS

David Young ⟨dyoung@NetBSD.org⟩

BUGS

To cope with interference from microwave ovens, frequency-hopping radios, and other sources of RF pulse-trains and bursts, **rssadapt** should adapt the fragmentation threshold as well as the data rate.

For improved throughput, **rssadapt** should indicate to drivers when they should use the 802.11b short-preamble.

The constants in **ieee80211_rssadapt_updatestats**() should be configurable.

NAME

rt_timer, **rt_timer_add**, **rt_timer_queue_create**, **rt_timer_queue_change**, **rt_timer_queue_destroy**, **rt_timer_remove_all** — route callout functions

SYNOPSIS

```
#include <net/route.h>

struct rttimer_queue *
rt_timer_queue_create(time_t timeout);

void
rt_timer_queue_change(struct rttimer_queue *q, time_t timeout);

void
rt_timer_queue_destroy(struct rttimer_queue *q, int destroy);

int
rt_timer_add(struct rtentry *rt,
    void(*f)(struct rtentry *, struct rttimer *),
    struct rttimer_queue *q);

void
rt_timer_remove_all(struct rtentry *rt);
```

DESCRIPTION

The **rt_timer** functions provide a generic route callout functionality. They allow a function to be called for a route at any time. This was originally intended to be used to remove routes added by path MTU discovery code.

For maximum efficiency, a separate queue should be defined for each timeout period. For example, one queue should be created for the 10 minute path MTU discovery timeouts, another for 20 minute ARP timeouts after 20 minutes, and so on. This permits extremely fast queue manipulations so that the timeout functions remain scalable, even in the face of thousands of route manipulations per minute.

It is possible to create only a single timeout queue for all possible timeout values, but doing so is not scalable as queue manipulations become quite expensive if the timeout deltas are not roughly constant.

The **rt_timer** interface provides the following functions:

rt_timer_queue_create(*time_t timeout*)
: This function creates a new timer queue with the specified timeout period *timeout*, expressed in seconds.

rt_timer_queue_change(*rttimer_queue *q*, *time_t timeout*)
: This function modifies the timeout period for a timer queue. Any value, including 0, is valid. The next time the timer queue's timeout expires (based on the previous timeout value), all entries which are valid to execute based on the new timeout will be executed, and the new timeout period scheduled.

rt_timer_queue_destroy(*rttimer_queue *q*, *int destroy*)
: This function destroys a timeout queue. All entries are removed, and if the *destroy* argument is non-zero, the timeout action is performed for each entry.

rt_timer_add(*struct rtentry *rt*, *void(*f)(struct rtentry *, struct rttimer *)*, *struct rttimer_queue *q*)
: This function adds an entry to a timeout queue. The function *f* will be called after the timeout period for queue *q* has elapsed. If *f* is NULL the route will be deleted when the timeout expires.

rt_timer_remove_all(*struct rtentry *rt*)
This function removes all references to the given route from the **rt_timer** subsystem. This is used when a route is deleted to ensure that no dangling references remain.

SEE ALSO
`netstat`(1), `arp`(9)

AUTHORS
This interface is roughly based on (but, alas, not compatible with) one designed by David Borman of BSDI. This implementation is by Kevin Lahey of the Numerical Aerospace Simulation Facility, NASA Ames Research Center.

CODE REFERENCES
The **rt_timer** interface is implemented in `sys/net/route.h` and `sys/net/route.c`.

HISTORY
The **rt_timer** interface appeared in NetBSD 1.4.

NAME

rw, **rw_init**, **rw_destroy**, **rw_enter**, **rw_exit**, **rw_tryenter**, **rw_tryupgrade**, **rw_downgrade**, **rw_read_held**, **rw_write_held**, **rw_lock_held** — reader / writer lock primitives

SYNOPSIS

```
#include <sys/rwlock.h>

void
rw_init(krwlock_t *rw);

void
rw_destroy(krwlock_t *rw);

void
rw_enter(krwlock_t *rw, const krw_t op);

void
rw_exit(krwlock_t *rw);

int
rw_tryenter(krwlock_t *rw, const krw_t op);

int
rw_tryupgrade(krwlock_t *rw);

void
rw_downgrade(krwlock_t *rw);

int
rw_read_held(krwlock_t *rw);

int
rw_write_held(krwlock_t *rw);

int
rw_lock_held(krwlock_t *rw);

options DIAGNOSTIC
options LOCKDEBUG
```

DESCRIPTION

Reader / writer locks (RW locks) are used in the kernel to synchronize access to an object among LWPs (lightweight processes) and soft interrupt handlers.

In addition to the capabilities provided by mutexes, RW locks distinguish between read (shared) and write (exclusive) access. RW locks are intended to provide protection for kernel data or objects that are read much more frequently than updated. For objects that are updated as frequently as they are read, mutexes should be used to guarantee atomic access.

RW locks are in one of three distinct states at any given time:

`Unlocked` The lock is not held.

`Read locked` The lock holders intend to read the protected object. Multiple callers may hold a RW lock with "read intent" simultaneously.

`Write locked` The lock holder intends to update the protected object. Only one caller may hold a RW lock with "write intent".

The *krwlock_t* type provides storage for the RW lock object. This should be treated as an opaque object and not examined directly by consumers.

Note that the these interfaces must not be used from a hardware interrupt handler.

OPTIONS AND MACROS

options DIANOSTIC

Kernels compiled with the DIAGNOSTIC option perform basic sanity checks on RW lock operations.

options LOCKDEBUG

Kernels compiled with the LOCKDEBUG option perform potentially CPU intensive sanity checks on RW lock operations.

FUNCTIONS

rw_init(*rw*)

Initialize a lock for use. No other operations can be performed on the lock until it has been initialized.

rw_destroy(*rw*)

Release resources used by a lock. The lock may not be used after it has been destroyed.

rw_enter(*rw*, *op*)

If RW_READER is specified as the argument to *op*, acquire a read lock. If the lock is write held, the caller will block and not return until the hold is acquired. Callers must not recursively acquire read locks.

If RW_WRITER is specified, acquire a write lock. If the lock is already held, the caller will block and not return until the hold is acquired.

RW locks and other types of locks must always be acquired in a consistent order with respect to each other. Otherwise, the potential for system deadlock exists.

rw_exit(*rw*)

Release a lock. The lock must have been previously acquired by the caller.

rw_tryenter(*rw*, *op*)

Try to acquire a lock, but do not block if the lock is already held. If the lock is acquired successfully, return non-zero. Otherwise, return zero.

Valid arguments to *op* are RW_READER or RW_WRITER.

rw_tryupgrade(*rw*)

Try to upgrade a lock from one read hold to a write hold. If the lock is upgraded successfully, returns non-zero. Otherwise, returns zero.

rw_downgrade(*rw*)

Downgrade a lock from a write hold to a read hold.

rw_write_held(*rw*)

rw_read_held(*rw*)

rw_lock_held(*rw*)

Test the lock's condition and return non-zero if the lock is held (potentially by the current LWP) and matches the specified condition. Otherwise, return zero.

These functions must never be used to make locking decisions at run time: they are provided only for diagnostic purposes.

CODE REFERENCES

This section describes places within the NetBSD source tree where code implementing RW locks can be found. All pathnames are relative to `/usr/src`.

The core of the RW lock implementation is in `sys/kern/kern_rwlock.c`.

The header file `sys/sys/rwlock.h` describes the public interface, and interfaces that machine-dependent code must provide to support RW locks.

SEE ALSO

`condvar`(9), `mb`(9), `mutex`(9)

Jim Mauro and Richard McDougall, *Solaris Internals: Core Kernel Architecture, Prentice Hall*, 2001, ISBN 0-13-022496-0.

HISTORY

The RW lock primitives first appeared in NetBSD 5.0.

NAME

sched_4bsd — The 4.4BSD thread scheduler

SYNOPSIS

```
#include <sys/sched.h>

void
resetpriority(lwp_t *l);

void
sched_tick(struct cpu_info *ci);

void
sched_schedclock(lwp_t *l);

void
sched_pstats_hook(struct proc *p, int minslp);

void
sched_setrunnable(lwp_t *l);

void
updatepri(lwp_t *l);
```

DESCRIPTION

The NetBSD thread scheduling sub-system employs a "multilevel feedback queues" algorithm, favouring interactive, short-running threads to CPU-bound ones.

resetpriority() recomputes the priority of a thread running in user mode. If the resulting priority is higher than that of the current thread, a reschedule is arranged.

sched_tick() gets called from hardclock(9) every 100ms to force a switch between equal priority threads.

The priority of the current thread is adjusted through **sched_schedclock**(). The priority of a thread gets worse as it accumulates CPU time.

sched_pstats_hook() gets called from **sched_pstats**() every Hz ticks in order to recompute the priorities of all threads.

sched_setrunnable() checks if an LWP has slept for more than one second. If so, its priority is updated by **updatepri**().

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing the scheduler can be found. All pathnames are relative to /usr/src.

The 4.4BSD scheduler subsystem is implemented within the file sys/kern/sched_4bsd.c.

SEE ALSO

csf(9), hardclock(9), mi_switch(9), userret(9)

Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, *Addison Wesley*, 1996.

NAME

scsipi — SCSI/ATAPI middle-layer interface

SYNOPSIS

```
#include <dev/scsipi/atapiconf.h>
#include <dev/scsipi/scsiconf.h>

void
scsipi_async_event(struct scsipi_channel *chan,
    scsipi_async_event_t event, void *arg);

void
scsipi_channel_freeze(struct scsipi_channel *chan, int count);

void
scsipi_channel_thaw(struct scsipi_channel *chan, int count);

void
scsipi_channel_timed_thaw(void *arg);

void
scsipi_periph_freeze(struct scsipi_periph *periph, int count);

void
scsipi_periph_thaw(struct scsipi_periph *periph, int count);

void
scsipi_periph_timed_thaw(void *arg);

void
scsipi_done(struct scsipi_xfer *xs);

void
scsipi_printaddr(struct scsipi_periph *periph);

int
scsipi_target_detach(struct scsipi_channel *chan, int target, int lun,
    int flags);

int
scsipi_thread_call_callback(struct scsipi_channel *chan,
    void (*callback)(struct scsipi_channel *, void *), void *arg);
```

DESCRIPTION

The **scsipi** system is the middle layer interface between SCSI/ATAPI host bus adapters (HBA) and high-level SCSI/ATAPI drivers. This document describes the interfaces provided by the **scsipi** layer towards the HBA layer. An HBA has to provide a pointer to a *struct scsipi_adapter* and one pointer per channel to a *struct scsipi_channel*. Once the SCSI or ATAPI bus is attached, the **scsipi** system will scan the bus and allocate a *struct scsipi_periph* for each device found on the bus. A high-level command (command sent from the high-level SCSI/ATAPI layer to the low-level HBA layer) is described by a *struct scsipi_xfer*.

A request is sent to the HBA driver though the **adapt_request**() callback. The HBA driver signals completion (with or without errors) of the request though **scsipi_done**(). **scsipi** knows the resources limits of the HBA (max number of concurrent requests per adapter of channel, and per periph), and will make sure the HBA won't receive more requests than it can handle.

The mid-layer can also handle `QUEUE FULL` and `CHECK CONDITION` events.

INITIALISATION

An HBA driver has to allocate and initialize to 0 a *struct scsipi_adapter* and fill in the following members:

*struct device *adapt_dev* — pointer to the HBA's *struct device*

int adapt_nchannels — number of channels (or busses) of the adapter

int adapt_openings — total number of commands the adapter can handle (may be replaced by *chan_openings*, see below)

int adapt_max_periph — number of commands the adapter can handle per device

The following callbacks should be provided through the *struct scsipi_adapter*:

void **(*adapt_request)**(*struct scsipi_channel **, *scsipi_adapter_req_t*, *void **)
mandatory

void **(*adapt_minphys)**(*struct buf **)
mandatory

int **(*adapt_ioctl)**(*struct scsipi_channel **, *u_long*, *void **, *int*, *struct lwp **)
optional

int **(*adapt_enable)**(*struct device **, *int*)
optional, set to NULL if not used

int **(*adapt_getgeom)**(*struct scsipi_periph **, *struct disk_parms **, *u_long*)
optional, set to NULL if not used

int **(*adapt_accesschk)**(*struct scsipi_periph **, *struct scsipi_inquiry_pattern **)
optional, set to NULL if not used

The HBA driver has to allocate and initialize to 0 one *struct scsipi_channel* per channel and fill in the following members:

*struct scsipi_adapter *chan_adapter*
Pointer to the HBA's *struct scsipi_adapter*

*struct scsipi_bustype *chan_bustype*
should be initialized to either *bus_atapi* or *bus_scsi*, both defined in the **scsipi** code.

int chan_channel — channel number (starting at 0)

int chan_flags — channel flags:

- SCSIPI_CHAN_OPENINGS — Use per-channel max number of commands *chan_openings* instead of per-adapter *adapt_openings*
- SCSIPI_CHAN_CANGROW — This channel can grow its *chan_openings* or *adapt_openings* on request (via the **adapt_request**() callback)
- SCSIPI_CHAN_NOSETTLE — Do not wait SCSI_DELAY seconds for devices to settle before probing (usually used by adapters that provide an "abstracted" view of the bus).

int chan_openings — total number of commands the adapter can handle for this channel (used only if the SCSIPI_CHAN_OPENINGS flag is set)

chan_max_periph — number of commands per device the adapter can handle on this channel (used only if the *SCSIPI_CHAN_OPENINGS* flag is set)

int chan_ntargets number of targets
int chan_nluns number of LUNs per target
int chan_id adapter's ID on this channel
int chan_defquirks default device quirks. Quirks are defined in <dev/scsipi/scsipiconf.h> and are usually set in the middle layer based on the device's inquiry data. For some kinds of adapters it may be convenient to have a set of quirks applied to all devices, regardless of the inquiry data.

The HBA driver attaches the SCSI or ATAPI bus (depending on the setting of *chan_bustype*) by passing a pointer to the *struct scsipi_channel* to the autoconf(4) machinery. The print function shall be either **scsiprint**() or **atapiprint**().

OTHER DATA STRUCTURES

When scanning the bus, the **scsipi** system allocates a *struct scsipi_periph* for each device probed. The interesting fields are:

*struct device *periph_dev* pointer to the device's *struct device*
*struct scsipi_channel *periph_channel*
pointer to the channel the device is connected to
int periph_quirks device quirks, defined in ⟨dev/scsipi/scsipiconf.h⟩
int periph_target target ID, or drive number on ATAPI
int periph_lun LUN (currently not used on ATAPI)

A SCSI or ATAPI request is passed to the HBA through a *struct scsipi_xfer*. The HBA driver has access to the following data:

struct callout xs_callout
callout for adapter use, usually for command timeout
int xs_control control flags (only flags of interest for HBA drivers are described):
XS_CTL_POLL poll in the HBA driver for request completion (most likely because interrupts are disabled)
XS_CTL_RESET reset the device
XS_CTL_DATA_UIO xs_data points to a *struct uio* buffer
XS_CTL_DATA_IN data is transferred from HBA to memory
XS_CTL_DATA_OUT data is transferred from memory to HBA
XS_CTL_DISCOVERY this xfer is part of a device discovery done by the middle layer
XS_CTL_REQSENSE xfer is a request sense

int xs_status status flags:
XS_STS_DONE xfer is done (set by **scsipi_done**())
XS_STS_PRIVATE mask of flags reserved for HBA's use (0xf0000000)

*struct scsipi_periph *xs_periph*
periph doing the xfer
int timeout command timeout, in milliseconds. The HBA should start the timeout at the time the command is accepted by the device. If the timeout happens, the HBA shall terminate the command through **scsipi_done**() with a XS_TIMEOUT error
*struct scsipi_generic *cmd*
scsipi command to execute
int cmdlen len (in bytes) of the cmd buffer

*u_char *data* data buffer (this is either a DMA or uio address)
int datalen data length (in bytes, zero if uio)
int resid difference between `datalen` and how much data was really transferred
scsipi_xfer_result_t error
error value returned by the HBA driver to mid-layer. See description of **scsipi_done**() for valid values
union {struct scsipi_sense_data scsi_sense; uint32_t atapi_sense;} sense
where to store sense info if `error` is XS_SENSE or XS_SHORTSENSE
uint8_t status SCSI status; checked by middle layer when `error is` XS_BUSY (the middle layer handles SCSI_CHECK and SCSI_QUEUE_FULL)
uint8_t xs_tag_type SCSI tag type, set to 0 if untagged command
uint8_t xs_tag_id tag ID, used for tagged commands

FUNCTIONS AND CALLBACKS

(*adapt_request)(*struct scsipi_channel *chan, scsipi_adapter_req_t req, void *arg*)
Used by the mid-layer to transmit a request to the adapter. *req* can be one of:
ADAPTER_REQ_RUN_XFER
request the adapter to send a command to the device. `arg` is a pointer to the *struct scsipi_xfer*. Once the xfer is complete the HBA driver shall call **scsipi_done**() with updated status and error information.
ADAPTER_REQ_GROW_RESOURCES
ask the adapter to increase resources of the channel (grow *adapt_openings* or *chan_openings*) if possible. Support of this feature is optional. This request is called from the kernel completion thread. `arg` must be ignored.
ADAPTER_REQ_SET_XFER_MODE
set the xfer mode for a for I_T Nexus. This will be called once all LUNs of a target have been probed. `arg` points to a *struct scsipi_xfer_mode* defined as follows:
int xm_target target for I_T Nexus
int xm_mode bitmask of device capabilities
int xm_period sync period
int xm_offset sync offset

xm_period and *xm_offset* shall be ignored for ADAPTER_REQ_SET_XFER_MODE. *xm_mode* holds the following bits:
PERIPH_CAP_SYNC
ST synchronous transfers
PERIPH_CAP_WIDE16
ST 16 bit wide transfers
PERIPH_CAP_WIDE32
ST 32 bit wide transfers
PERIPH_CAP_DT
DT transfers
PERIPH_CAP_TQING
tagged queueing
Whenever the xfer mode changes, the driver should call **scsipi_async_event**() to notify the mid-layer.

adapt_request() may be called from interrupt context.

adapt_minphys()
pointer to the driver's minphys function. If the driver can handle transfers of size MAXPHYS, this can point to **minphys**().

adapt_ioctl()
ioctl function for the channel. The only ioctl supported at this level is SCBUSIORESET for which the HBA driver shall issue a SCSI reset on the channel.

int **adapt_enable**(*struct device *dev*, *int enable*)
Disable the adapter if *enable* is zero, or enable it if non-zero. Returns 0 if operation is successful, or error from <sys/errno.h>. This callback is optional, and is useful mostly for hotplug devices. For example, this callback would power on or off the relevant PCMCIA socket for a PCMCIA controller.

int **adapt_getgeom**(*struct scsipi_periph *periph*, *struct disk_parms *params*, *u_long sectors*)
Optional callback, used by high-level drivers to get the fictitious geometry used by the controller's firmware for the specified periph. Returns 0 if successful. See Adaptec drivers for details.

int **adapt_accesschk**(*struct scsipi_periph *periph*, *struct scsipi_inquiry_pattern *inqbuf*)
Optional callback; if present the mid-layer uses it to check if it can attach a driver to the specified periph. If the callback returns a non-zero value, the periph is ignored by the **scsipi** code. This callback is used by adapters which want to drive some devices themselves, for example hardware RAID controllers.

scsipi_async_event(*struct scsipi_channel *chan*, *scsipi_async_event_t event*, *void *arg*)
Asynchronous event notification for the mid-layer. *event* can be one of:

ASYNC_EVENT_MAX_OPENINGS
set max openings for a periph. Argument is a *struct scsipi_max_openings* with at least the following members:
int mo_target
int mo_lun
int mo_openings

Not all periphs may allow openings to increase; if not allowed the request is silently ignored.

ASYNC_EVENT_XFER_MODE
update the xfer mode for an I_T nexus. Argument is a *struct scsipi_xfer_mode* properly filled in. An ASYNC_EVENT_XFER_MODE call with PERIPH_CAP_TQING set in *xm_mode* is mandatory to activate tagged queuing.

ASYNC_EVENT_RESET
channel has been reset. No argument. HBA drivers have to issue ASYNC_EVENT_RESET events if they rely on the mid-layer for SCSI CHECK CONDITION handling.

scsipi_done(*struct scsipi_xfer *xs*)
shall be called by the HBA when the xfer is complete, or when it needs to be requeued by the mid-layer. *error* in the scsipi_xfer shall be set to one of the following:

XS_NOERROR
xfer completed without error.

XS_SENSE Check the returned SCSI sense for the error.

XS_SHORTSENSE
Check the ATAPI sense for the error.

XS_DRIVER_STUFFUP
Driver failed to perform operation.

XS_RESOURCE_SHORTAGE
: Adapter resource shortage. The mid-layer will retry the command after some delay.

XS_SELTIMEOUT
: The device timed out while trying to send the command

XS_TIMEOUT
: The command was accepted by the device, but it didn't complete in allowed time.

XS_BUSY
: The mid-layer will check *status* for additional details:

  SCSI_CHECK
  : SCSI check condition. The mid-layer will freeze the periph queue and issue a REQUEST SENSE command. If the HBA supports tagged queuing, it shall remove and requeue any command not yet accepted by the HBA (or at last make sure no more commands will be sent to the device before the REQUEST SENSE is complete).

  SCSI_QUEUE_FULL
  : The mid layer will adjust the periph's openings and requeue the command.

  SCSI_BUSY
  : The mid-layer will requeue the xfer after delay.

XS_RESET
: xfer destroyed by a reset; the mid-layer will requeue it.

XS_REQUEUE
: Ask the mid-layer to requeue this command immediately.

The adapter should not reference an *xfer* once **scsipi_done**(*xfer*) has been called, unless the *xfer* had XS_CTL_POLL set.

scsipi_done() will call the **adapt_request**() callback again only if called with *xs->error* set to XS_NOERROR, and *xfer* doesn't have XS_CTL_POLL set. All other error conditions are handled by a kernel thread (once the HBA's interrupt handler has returned).

scsipi_printaddr(*struct scsipi_periph *periph*)
: print a kernel message with the periph's name, in the form device(controller:channel:target:lun).

scsipi_channel_freeze(*struct scsipi_channel *chan*, *int count*)
: Freeze the specified channel (requests are queued but not sent to HBA). The channel's freeze counter is increased by *count*.

scsipi_channel_thaw(*struct scsipi_channel *chan*, *int count*)
: Decrement the channel's freeze counter by *count* and process the queue if the counter goes to 0. In order to preserve command ordering, HBA drivers should not call **scsipi_channel_thaw**() before calling **scsipi_done**() for all commands in the HBA's queue which need to be requeued.

scsipi_periph_timed_thaw(*void *arg*)
: Call **scsipi_channel_thaw**(*arg*, *1*). Intended to be used as callout(9) callback.

scsipi_periph_freeze(*struct scsipi_periph *periph*, *int count*)

scsipi_periph_thaw(*struct scsipi_periph *periph*)

scsipi_periph_timed_thaw(*void *arg*)
: Same as the channel counterparts, but only for one specific peripheral.

scsipi_target_detach(*struct scsipi_channel *chan*, *int target*, *int lun*, *int flags*)
: detach the periph associated with this I_T_L nexus. Both *target* and *lun* may be wildcarded using the magic value -1. *flags* is passed to **config_detach**() . Returns 0 if successful, or error code if a device couldn't be removed.

scsipi_thread_call_callback(*struct scsipi_channel *chan*, *void (*callback)(struct scsipi_channel *, void *)*, *void *arg*)
: **callback**() will be called with *chan* and *arg* as arguments, from the channel completion thread. The callback is run at splbio. **scsipi_thread_call_callback**() will freeze the channel by one, it's up to the caller to thaw it when appropriate. Returns 0 if the callback was

properly recorded, or EBUSY if the channel has already a callback pending.

FILES

`sys/dev/scsiconf.h` header file for use by SCSI HBA drivers

`sys/dev/atapiconf.h` header file for use by ATAPI HBA drivers

Both header files include `sys/dev/scsipiconf.h` which contains most structure definitions, function prototypes and macros.

EXAMPLES

The best examples are existing HBA drivers. Most of them sit in the `sys/dev/ic` directory.

HISTORY

The **`scsipi`** interface appeared in NetBSD 1.6.

AUTHORS

The **`scsipi`** interface was designed and implemented by Jason R. Thorpe. Manuel Bouyer converted most drivers to the new interface.

NAME

secmodel — security model development guidelines

SYNOPSIS

```
#include <secmodel/secmodel.h>
```

DESCRIPTION

NetBSD provides a complete abstraction of the underlying security model used with the operating system to a set of kauth(9) scopes and actions.

It is possible to modify the security model -- either slightly or using an entirely different model -- by attaching/detaching kauth(9) listeners. This document describes this process.

Background

In NetBSD 4.0, Kernel Authorization -- kauth(9) -- was introduced as the subsystem responsible for authorization and credential management. Before its introduction, there were several ways for providing resource access control:

- Checking if the user in question is the superuser via **suser**().
- Comparing the user-id against hard-coded values, often zero,
- Checking the system securelevel.

The problem with the above is that the interface ("can X do Y?") was tightly coupled with the implementation ("is X Z?"). kauth(9) allowed us to separate them, dispatching requests with highly detailed context using a consistent and clear KPI.

The result is a pluggable framework for attaching "listeners" that can modify the behavior of the system, security-wise. It allows us to maintain the existing security model (based on a single superuser and above-superuser restrictions known as securelevel) but easily decouple it from the system, given we want to use a different one.

The different security model can be implemented in the kernel or loaded as an LKM, base its decisions on available information, dispatch the decision to a userspace daemon, or even to a centralized network authorization server.

The kauth(9) KPI

Before writing a new security model, one should be familiar with the kauth(9) KPI, its limitations, requirements, and so on.

First, some terminology. According to kauth(9), the system is logically divided to scopes, where each scope denotes a different area of interest in the system -- something like a namespace. For example, NetBSD has the process, network, and machdep scopes, representing process-related, network-related, and machdep-related actions.

Each scope has a collection of actions -- or requests -- forming the high level indication of the request type. Each request is automatically associated with credentials and between zero to four arguments providing the request context.

For example, in the process scope there are requests such as "can signal", "can change rlimits", and "can change corename".

Each scope in the system is associated with listeners, which are actually callback routines, that get called when an authorization request on the relevant scope takes place.

Every listener receives the request and its context, and can make a decision of either "allow", "deny", or "defer" (if it doesn't want to be the one deciding).

It is important to note that a single "deny" is enough to fail a request, and at least a single "allow" is required to allow it. In other words, it is impossible to attach listeners that weaken the security of the system or override decisions made by other listeners.

At last, there are several things you should remember about kauth(9):

- Authorization requests can not be issued when the kernel is holding any locks. This is a requirement from kernel code, to allow designing security models where the request should be dispatched to userspace or a different host.
- Private listener data -- such as internal data-structures -- is entirely under the responsibility of the developer. Locking, synchronization, and garbage collection are all things that kauth(9) does *not* take care of for you!

Writing a new security model

A security model is composed of (code-wise) the following components:

1. Entry routines, named **secmodel_<model>_init**() and **secmodel_<model>_start**(), used to initialize and start the security model.

   If the security model is to be started automatically by the kernel and is compiled in it, a function called **secmodel_start**() can be added to call the model's start routine.

   If the security model is to be built and used as an LKM, another function called **secmodel_<model>_stop**(), to stop the security model in case the module is to be unloaded.

2. A sysctl(9) setup routine for the model. This should create an entry for the model in the sysctl(9) namespace, under the "security.models.<model>" hierarchy.

   All "knobs" for the model should be located under the new node, as well as a mandatory "name" variable, indicating a descriptive human-readable name for the model.

   If the module is to be used as an LKM, explicit calls to the setup routine and **sysctl_teardown**() are to be used to create and destroy the sysctl(9) tree.

3. If the model uses any private data inside credentials, listening on the credentials scope, KAUTH_SCOPE_CRED, is required.

4. Optionally, internal data-structures used by the model. These must all be prefixed with "secmodel_<model>_".

5. A set of listeners, attached to various scopes, used to enforce the policy the model intends to implement.

6. Finally, a security model should register itself when loaded using **secmodel_register**() and deregister it when unloaded (if used as an LKM) using **secmodel_deregister**().

Below is sample code for a kauth(9) network scope listener for the *jenna* security model. It is used to allow users with a user-id below 1000 bind to reserved ports (for example, 22/TCP):

```
int
secmodel_jenna_network_cb(kauth_cred_t cred, kauth_action_t action,
    void *cookie, void *arg0, void *arg1, void *arg2, void *arg3)
{
        int result;

        /* Default defer. */
        result = KAUTH_RESULT_DEFER;
```

```
        switch (action) {
        case KAUTH_NETWORK_BIND:
                /*
                 * We only care about bind(2) requests to privileged
                 * ports.
                 */
                if ((u_long)arg0 == KAUTH_REQ_NETWORK_BIND_PRIVPORT) {
                        /*
                         * If the user-id is below 1000, which may
                         * indicate a "reserved" user-id, allow the
                         * request.
                         */
                        if (kauth_cred_geteuid(cred) < 1000)
                                result = KAUTH_RESULT_ALLOW;
                }
                break;
        }

        return (result);
}
```

There are two main issues, however, with that listener, that you should be aware of when approaching to write your own security model:

1. As mentioned, kauth(9) uses restrictive decisions: if you attach this listener on-top of an existing security model, even if it would allow the request, it could still be failed.
2. If you attach this listener as the only listener for the network scope, there are many other requests that will be deferred and, eventually, denied -- which may not be desired.

That's why before implementing listeners, it should be clear whether they implement an entirely new from scratch security model, or add on-top of an existing one.

Adding on-top of an existing security model

One of the shortcomings of kauth(9) is that it does not provide any stacking mechanism, similar to Linux Security Modules (LSM). This, however, is considered a feature in reducing dependency on other people's code.

To properly "stack" minor adjustments on-top of an existing security model, one could use one of two approaches:

- Registering an internal scope for the security model to be used as a fall-back when requests are deferred.

  This requires the security model developer to add an internal scope for every scope the model partly covers, and registering the fall-back listeners to it. In the model's listener(s) for the scope, when a defer decision is made, the request is passed to be authorized on the internal scope, effectively using the fall-back security model.

  Here's example code that implements the above:

```
#include <secmodel/bsd44/bsd44.h>

/*
 * Internal fall-back scope for the network scope.
 */
#define JENNA_ISCOPE_NETWORK "jenna.iscope.network"
```

```
static kauth_scope_t secmodel_jenna_iscope_network;

/*
 * Jenna's entry point. Register internal scope for the network scope
 * which we partly cover for fall-back authorization.
 */
void
secmodel_jenna_start(void)
{
        secmodel_jenna_iscope_network = kauth_register_scope(
            JENNA_ISCOPE_NETWORK, NULL, NULL);

        kauth_listen_scope(JENNA_ISCOPE_NETWORK,
            secmodel_bsd44_suser_network_cb, NULL);
        kauth_listen_scope(JENNA_ISCOPE_NETWORK,
            secmodel_bsd44_securelevel_network_cb, NULL);
}

/*
 * Jenna sits on top of another model, effectively filtering requests.
 * If it has nothing to say, it discards the request. This is a good
 * example for fine-tuning a security model for a special need.
 */
int
secmodel_jenna_network_cb(kauth_cred_t cred, kauth_action_t action,
    void *cookie, void *arg0, void *arg1, void *arg2, void *arg3)
{
        int result;

        /* Default defer. */
        result = KAUTH_RESULT_DEFER;

        switch (action) {
        case KAUTH_NETWORK_BIND:
                /*
                 * We only care about bind(2) requests to privileged
                 * ports.
                 */
                if ((u_long)arg0 == KAUTH_REQ_NETWORK_BIND_PRIVPORT) {
                        if (kauth_cred_geteuid(cred) < 1000)
                                result = KAUTH_RESULT_ALLOW;
                }
                break;
        }

        /*
         * If we have don't have a decision, fall-back to the bsd44
         * security model.
         */
        if (result == KAUTH_RESULT_DEFER)
                result = kauth_authorize_action(
                    secmodel_jenna_iscope_network, cred, action,
```

```
                        arg0, arg1, arg2, arg3);

        return (result);
}
```

- If the above is not desired, or cannot be used for any reason, there is always the ability to manually call the fall-back routine:

```
int
secmodel_jenna_network_cb(kauth_cred_t cred, kauth_action_t action,
    void *cookie, void *arg0, void *arg1, void *arg2, void *arg3)
{
        int result;

        /* Default defer. */
        result = KAUTH_RESULT_DEFER;

        switch (action) {
        case KAUTH_NETWORK_BIND:
                /*
                 * We only care about bind(2) requests to privileged
                 * ports.
                 */
                if ((u_long)arg0 == KAUTH_REQ_NETWORK_BIND_PRIVPORT) {
                        if (kauth_cred_geteuid(cred) < 1000)
                                result = KAUTH_RESULT_ALLOW;
                }
                break;
        }

        /*
         * If we have don't have a decision, fall-back to the bsd44
         * security model's suser behavior.
         */
        if (result == KAUTH_RESULT_DEFER)
                result = secmodel_bsd44_suser_network_cb(cred, action,
                    cookie, arg0, arg1, arg2, arg3);

        return (result);
}
```

Writing a new security model from scratch

When writing a security model from scratch, aside from the obvious issues of carefully following the desired policy to be implemented and paying attention to all of the issues outlined above, one must also remember that any unhandled requests will be denied by default.

To make it easier on developers to write new security models from scratch, NetBSD maintains skeleton listeners that contain every possible request and arguments.

Available security models

The following is a list of security models available in the default NetBSD distribution. To choose, one should edit `/usr/src/sys/conf/std`.

secmodel_bsd44
: Traditional NetBSD security model, derived from 4.4BSD.

secmodel_overlay
: Sample overlay security model, sitting on-top of `secmodel_bsd44`(9).

FILES

`/usr/share/examples/secmodel`

SEE ALSO

`kauth`(9), `secmodel_bsd44`(9), `secmodel_overlay`(9)

AUTHORS

Elad Efrat ⟨elad@NetBSD.org⟩

NAME

secmodel_bsd44 — traditional NetBSD security model (based on 4.4BSD)

DESCRIPTION

secmodel_bsd44 is the default security model in NetBSD. It is the traditional security model based on 4.4BSD and is composed of two main concepts, the *super-user* and the *securelevel*.

Super-user

The *super-user* is the host administrator, considered to have higher privileges than other users. It is the only entity the kernel recognizes by having an effective user-id of zero.

Securelevel

Please refer to secmodel_securelevel(9) for details.

SEE ALSO

kauth(9), secmodel(9), secmodel_securelevel(9)

AUTHORS

Elad Efrat ⟨elad@NetBSD.org⟩

NAME

secmodel_overlay — sample overlay security model implementation

SYNOPSIS

```
#include <secmodel/overlay/overlay.h>
```

DESCRIPTION

secmodel_overlay is a sample implementation for an overlay security model. It can be thought of as a "filter" for the underlying model it overlays, by default it is secmodel_bsd44(9), where developers or administrators can implement custom policies using least intrusive code changes.

FILES

/usr/src/sys/secmodel/overlay/secmodel_overlay.c

SEE ALSO

kauth(9), secmodel(9), secmodel_bsd44(9)

AUTHORS

Elad Efrat ⟨elad@NetBSD.org⟩

NAME

secmodel_bsd44 — traditional NetBSD security model (based on 4.4BSD)

DESCRIPTION

The securelevel mechanism is intended to allow protecting the persistence of code and data on the system, or a subset thereof, from modification, even by the super-user, by providing convenient means of "locking down" a system to a degree suited to its environment.

The super-user can raise the securelevel using `sysctl`(8), but only `init`(8) can lower it.

secmodel_bsd44 provides four levels of securelevel, defined as follows:

-1 *Permanently insecure mode*

- Don't raise the securelevel on boot

0 *Insecure mode*

- The init process (PID 1) may not be traced or accessed by `ptrace`(2) or procfs.
- Immutable and append-only file flags may be changed
- All devices may be read or written subject to their permissions

1 *Secure mode*

- All effects of securelevel 0
- `/dev/mem` and `/dev/kmem` may not be written to
- Raw disk devices of mounted file systems are read-only
- Immutable and append-only file flags may not be removed
- Kernel modules may not be loaded or unloaded
- The *net.inet.ip.sourceroute* `sysctl`(8) variable may not be changed
- Adding or removing `sysctl`(9) nodes is denied
- The RTC offset may not be changed
- Set-id coredump settings may not be altered
- Attaching the IP-based kernel debugger, `ipkdb`(4), is not allowed
- Device "pass-thru" requests that may be used to perform raw disk and/or memory access are denied
- *iopl* and *ioperm* calls are denied
- Access to unmanaged memory is denied

2 *Highly secure mode*

- All effects of securelevel 1
- Raw disk devices are always read-only whether mounted or not
- New disks may not be mounted, and existing mounts may only be downgraded from read-write to read-only
- The system clock may not be set backwards or close to overflow
- Per-process coredump name may not be changed
- Packet filtering and NAT rules may not be altered

Highly secure mode may seem Draconian, but is intended as a last line of defence should the superuser account be compromised. Its effects preclude circumvention of file flags by direct modification of a raw disk device, or erasure of a file system by means of `newfs`(8). Further, it can limit the potential damage of a compromised "firewall" by prohibiting the modification of packet filter rules. Preventing the system clock from being set backwards aids in post-mortem analysis and helps ensure the integrity of logs. Precision timekeeping is not affected because the clock may still be slowed.

Normally, the system runs in securelevel 0 while single-user and in securelevel 1 while multi-user. If a higher securelevel is desired while running multi-user, it can be set using the *securelevel* keyword in the startup script `/etc/rc.conf`, see `rc.conf`(5) for details. Lower securelevels require the kernel to be compiled with **options INSECURE**, causing it to always default to securelevel −1.

In order for this protection to be effective, the administrator must ensure that no program that is run while the security level is 0 or lower, nor any data or configuration file used by any such program, can be modified while the security level is greater than 0. This may be achieved through the careful use of the "immutable" file flag to define and protect a Trusted Computing Base (TCB) consisting of all such programs and data, or by ensuring that all such programs and data are on filesystems that are mounted read-only and running at security level 2 or higher. *Particular care must be taken to ensure, if relying upon security level 1 and the use of file flags, that the integrity of the TCB cannot be compromised through the use of modifications to the disklabel or access to overlapping disk partitions, including the raw partition.*

Do not overlook the fact that shell scripts (or anything else fed to an interpreter, through any mechanism) and the kernel itself are "programs that run while the security level is 0" and must be considered part of the TCB.

SEE ALSO

`kauth`(9), `secmodel`(9), `secmodel_bsd44`(9)

AUTHORS

Elad Efrat ⟨elad@NetBSD.org⟩

BUGS

Systems without `sysctl`(8) behave as though they have security level −1.

The security level 2 restrictions relating to TCB integrity protection should be enforced at security level 1. Restrictions dependent upon security level but not relating to TCB integrity protection should be selected by `sysctl`(8) settings available only at security level 0 or lower.

NAME

seldestroy, **selinit**, **selrecord**, **selnotify** — select and poll subsystem

SYNOPSIS

```
#include <sys/param.h>
#include <sys/select.h>

void
seldestroy(struct selinfo *sip);

void
selinit(struct selinfo *sip);

void
selrecord(struct lwp *selector, struct selinfo *sip);

void
selnotify(struct selinfo *sip, int events, long knhint);
```

DESCRIPTION

selinit() and **seldestroy**() functions must be used to initialize and destroy the *struct selinfo*. The **seldestroy**() function may block.

selrecord() and **selnotify**() are used by device drivers to coordinate with the kernel implementation of `select`(2) and `poll`(2). Each object that can be polled contains a `selinfo` record. Device drivers provide locking for the `selinfo` record.

selrecord() records that the calling thread is interested in events related to a given object. **selrecord**() should only be called when the poll routine determines that the object is not ready for I/O: there are no events of interest pending. The check for pending I/O and call to **selrecord**() must be atomic. Atomicity can be provided by holding the object's lock across the test and call to **selrecord**(). For non-MPSAFE drivers, the global `kernel_lock` is enough to provide atomicity.

selnotify() is called by the underlying object handling code in order to notify any waiting threads that an event of interest has occurred. The same lock held across the poll method and call to **selrecord**() must be held across the call to **selnotify**(). The lock prevents an event of interest being signalled while a thread is in the process of recording its interest.

The *events* indicates which event happen. Zero may be used if unknown.

selnotify() also calls **KNOTE**() passing *knhint* as an argument.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing select and poll subsystem can be found. All pathnames are relative to `/usr/src`.

The core of the select and poll subsystem implementation is in `sys/kern/sys_select.c`. Data structures and function prototypes are located in `sys/sys/select.h`, `sys/sys/poll.h` and `sys/sys/selinfo.h`.

SEE ALSO

`poll`(2), `select`(2), `knote`(9)

NAME

setjmp, **longjmp** — non-local jumps

SYNOPSIS

```
#include <machine/types.h>
#include <sys/systm.h>

int
setjmp(label_t *label);

void
longjmp(label_t *label);
```

DESCRIPTION

The **setjmp**() function saves its calling environment in *label*. It returns zero on success. The **longjmp**() function restores the environment saved by the most recent invocation of **setjmp**(). It returns so that kernel execution continues as if the corresponding invocation of the **setjmp**() had just returned.

setjmp() and **longjmp**() are a machine-independent interface for machine-dependent implementations.

These functions are primarily used by ddb(4).

SEE ALSO

ddb(4)

NAME

shutdownhook_establish, **shutdownhook_disestablish** — add or remove a shutdown hook

SYNOPSIS

void *
shutdownhook_establish(*void (*fn)(void *)*, *void *arg*);

void
shutdownhook_disestablish(*void *cookie*);

DESCRIPTION

The **shutdownhook_establish**() function adds *fn* to the list of hooks invoked by doshutdownhooks(9) at shutdown. When invoked, the hook function *fn* will be passed *arg* as its only argument.

The **shutdownhook_disestablish**() function removes the hook described by the opaque pointer *cookie* from the list of hooks to be invoked at shutdown. If *cookie* is invalid, the result of **shutdownhook_disestablish**() is undefined.

Shutdown hooks should be used to perform one-time activities that must happen immediately before the kernel exits. Because of the environment in which they are run, shutdown hooks cannot rely on many system services (including file systems, and timeouts and other interrupt-driven services), or even basic system integrity (because the system could be rebooting after a crash).

RETURN VALUES

If successful, **shutdownhook_establish**() returns an opaque pointer describing the newly-established shutdown hook. Otherwise, it returns NULL.

EXAMPLES

It may be appropriate to use a shutdown hook to disable a device that does direct memory access, so that the device will not try to access memory while the system is rebooting.

It may be appropriate to use a shutdown hook to inform watchdog timer hardware that the operating system is no longer running.

SEE ALSO

doshutdownhooks(9)

BUGS

The names are clumsy, at best.

NAME

signal, **siginit**, **sigactsinit**, **sigactsunshare**, **sigactsfree**, **execsigs**, **sigaction1**, **sigprocmask1**, **sigpending1**, **sigsuspend1**, **sigaltstack1**, **gsignal**, **pgsignal**, **psignal**, **sched_psignal**, **issignal**, **postsig**, **killproc**, **sigexit**, **sigmasked**, **trapsignal**, **sendsig**, **sigcode**, **sigtramp** — software signal facilities

SYNOPSIS

```
#include <sys/signal.h>
#include <sys/signalvar.h>

void
siginit(struct proc *p);

void
sigactsinit(struct proc *np, struct proc *pp, int share);

void
sigactsunsuare(struct proc *p);

void
sigactsfree(struct proc *p);

void
execsigs(struct proc *p);

int
sigaction1(struct proc *p, int signum, const struct sigaction *nsa,
    struct sigaction *osa, void *tramp, int vers);

int
sigprocmask1(struct proc *p, int how, const sigset_t *nss, sigset_t *oss);

void
sigpending1(struct proc *p, sigset_t *ss);

int
sigsuspend1(struct proc *p, const sigset_t *ss);

int
sigaltstack1(struct proc *p, const struct sigaltstack *nss,
    struct sigaltstack *oss);

void
gsignal(int pgid, int signum);

void
kgsignal(int pgid, ksiginfo_t *ks, void *data);

void
pgsignal(struct pgrp *pgrp, int signum, int checkctty);

void
kpgsignal(struct pgrp *pgrp, ksiginfo_t *ks, void *data, int checkctty);

void
psignal(struct proc *p, int signum);

void
kpsignal(struct proc *p, ksiginfo_t *ks, void *data);
```

void
sched_psignal(*struct proc *p*, *int signum*);

int
issignal(*struct lwp *l*);

void
postsig(*int signum*);

void
killproc(*struct proc *p*, *const char *why*);

void
sigexit(*struct proc *p*, *int signum*);

int
sigmasked(*struct proc *p*, *int signum*);

void
trapsignal(*struct proc *p*, *const ksiginfo_t *ks*);

void
sendsig(*const ksiginfo_t *ks*, *const sigset_t *mask*);

DESCRIPTION

The system defines a set of signals that may be delivered to a process. These functions implement the kernel portion of the signal facility.

Signal numbers used throughout the kernel signal facilities should always be within the range of [1-NSIG].

Most of the kernel's signal infrastructure is implemented in machine-independent code. Machine-dependent code provides support for invoking a process's signal handler, restoring context when the signal handler returns, generating signals when hardware traps occur, triggering the delivery of signals when a process is about to return from the kernel to userspace.

The signal state for a process is contained in *struct sigctx*. This includes the list of signals with delivery pending, information about the signal handler stack, the signal mask, and the address of the signal trampoline.

The registered signal handlers for a process are recorded in *struct sigacts*. This structure may be shared by multiple processes.

The kernel's signal facilities are implemented by the following functions:

void **siginit**(*struct proc *p*)

> This function initializes the signal state of *proc0* to the system default. This signal state is then inherited by `init`(8) when it is started by the kernel.

void **sigactsinit**(*struct proc *np*, *struct proc *pp*, *int share*)

> This function creates an initial *struct sigacts* for the process *np*. If the *share* argument is non-zero, then *np* shares the *struct sigacts* with the process *pp*. Otherwise, *np* receives a new *struct sigacts* which is copied from *pp* if non-`NULL`.

void **sigactsunshare**(*struct proc *p*)

> This function causes the process *p* to no longer share its *struct sigacts* The current state of the signal actions is maintained in the new copy.

void **sigactsfree**(*struct proc *p*)

This function decrements the reference count on the *struct sigacts* of process *p*. If the reference count reaches zero, the *struct sigacts* is freed.

void **execsigs**(*struct proc *p*)

This function is used to reset the signal state of the process *p* to the system defaults when the process execs a new program image.

int **sigaction1**(*struct proc *p*, *int signum*, *const struct sigaction *nsa*, *struct sigaction *osa*, *void *tramp*, *int vers*)

This function implements the sigaction(2) system call. The *tramp* and *vers* arguments provide support for userspace signal trampolines. Trampoline version 0 is reserved for the legacy kernel-provided signal trampoline; *tramp* must be NULL in this case. Otherwise, *vers* specifies the ABI of the trampoline specified by *tramp*. The signal trampoline ABI is machine-dependent, and must be coordinated with the **sendsig**() function.

int **sigprocmask1**(*struct proc *p*, *int how*, *const sigset_t *nss*, *sigset_t *oss*)

This function implements the sigprocmask(2) system call.

void **sigpending1**(*struct proc *p*, *sigset_t *ss*)

This function implements the sigpending(2) system call.

int **sigsuspend1**(*struct proc *p*, *const sigset_t *ss*)

This function implements the sigsuspend(2) system call.

int **sigaltstack1**(*struct proc *p*, *const struct sigaltstack *nss*, *struct sigaltstack *oss*)

This function implements the sigaltstack(2) system call.

void **gsignal**(*int pgid*, *int signum*)

This is a wrapper function for **kgsignal**() which is described below.

void **kgsignal**(*int pgid*, *ksiginfo_t *ks*, *void *data*)

Schedule the signal *ks->ksi_signo* to be delivered to all members of the process group specified by *pgid*. The *data* argument and the complete signal scheduling semantics are described in the **kpsignal**() function below. below for a complete description of the signal scheduling semantics.

void **pgsignal**(*struct pgrp *pgrp*, *int signum*, *int checkctty*)

This is a wrapper function for **kpgsignal**() which is described below.

void **kpgsignal**(*struct pgrp *pgrp*, *ksiginfo_t *ks*, *void *data*, *int checkctty*)

Schedule the signal *ks->ksi_signo* to be delivered to all members of the process group *pgrp*. If *checkctty* is non-zero, the signal is only sent to processes which have a controlling terminal. The *data* argument and the complete signal scheduling semantics are described in the **kpsignal**() function below.

void **trapsignal**(*struct proc *p*, *const ksiginfo_t *ks*)

Sends the signal *ks->ksi_signo* caused by a hardware trap to the process *p*. This function is meant to be called by machine-dependent trap handling code, through the p->p_emul->e_trapsignal function pointer because some emulations define their own

trapsignal functions that remap the signal information to what the emulation expects.

void **psignal**(*struct proc *p*, *int signum*)

This is a wrapper function for **kpgsignal**() which is described below.

void **kpsignal**(*struct proc *p*, *ksiginfo_t *ks*, *void *data*)

Schedule the signal *ks->ksi_signo* to be delivered to the process *p*. The *data* argument, if not NULL, points to the file descriptor data that caused the signal to be generated in the SIGIO case.

With a few exceptions noted below, the target process signal disposition is updated and is marked as runnable, so further handling of the signal is done in the context of the target process after a context switch; see **issignal**() below. Note that **kpsignal**() does not by itself cause a context switch to happen.

The target process is not marked as runnable in the following cases:

- The target process is sleeping uninterruptibly. The signal will be noticed when the process returns from the system call or trap.
- The target process is currently ignoring the signal.
- If a stop signal is sent to a sleeping process that takes the default action (see sigaction(2)), the process is stopped without awakening it.
- SIGCONT restarts a stopped process (or puts them back to sleep) regardless of the signal action (e.g., blocked or ignored).

If the target process is being traced, **kpsignal**() behaves as if the target process were taking the default action for *signum*. This allows the tracing process to be notified of the signal.

void **sched_psignal**(*struct proc *p*, *int signum*)

An alternate version of **kpsignal**() which is intended for use by code which holds the scheduler lock.

int **issignal**(*struct lwp *l*)

This function determines which signal, if any, is to be posted to the process *p*. A signal is to be posted if:

- The signal has a handler provided by the program image.
- The signal should cause the process to dump core and/or terminate.
- The signal should interrupt the current system call.

Signals which cause the process to be stopped are handled within **issignal**() directly.

issignal() should be called by machine-dependent code when returning to userspace from a system call or other trap or interrupt by using the following code:

```
while (signum = CURSIG(curproc))
        postsig(signum);
```

void **postsig**(*int signum*)

The **postsig**() function is used to invoke the action for the signal *signum* in the current process. If the default action of a signal is to terminate the process, and the signal does not have a registered handler, the process exits using **sigexit**(), dumping a core image if necessary.

void **killproc**(*struct proc *p*, *const char *why*)

This function sends a SIGKILL signal to the specified process. The message provided by *why* is sent to the system log and is also displayed on the process's controlling terminal.

void **sigexit**(*struct proc *p*, *int signum*)

This function forces the process *p* to exit with the signal *signum*, generating a core file if appropriate. No checks are made for masked or caught signals; the process always exits.

int **sigmasked**(*struct proc *p*, *int signum*)

This function returns non-zero if the signal specified by *signum* is ignored or masked for process *p*.

void **sendsig**(*const ksiginfo_t *ks*, *const sigset_t *mask*)

This function is provided by machine-dependent code, and is used to invoke a signal handler for the current process. **sendsig**() must prepare the registers and stack of the current process to invoke the signal handler stored in the process's *struct sigacts*. This may include switching to an alternate signal stack specified by the process. The previous register, stack, and signal state are stored in a *ucontext_t*, which is then copied out to the user's stack.

The registers and stack must be set up to invoke the signal handler as follows:

```
(*handler)(int signum, siginfo_t *info, void *ctx)
```

where *signum* is the signal number, *info* contains additional signal specific information when SA_SIGINFO is specified when setting up the signal handler. *ctx* is the pointer to *ucontext_t* on the user's stack. The registers and stack must also arrange for the signal handler to return to the signal trampoline. The trampoline is then used to return to the code which was executing when the signal was delivered using the setcontext(2) system call.

For performance reasons, it is recommended that **sendsig**() arrange for the signal handler to be invoked directly on architectures where it is convenient to do so. In this case, the trampoline is used only for the signal return path. If it is not feasible to directly invoke the signal handler, the trampoline is also used to invoke the handler, performing any final set up that was not possible for **sendsig**() to perform.

sendsig() must invoke the signal trampoline with the correct ABI. The ABI of the signal trampoline is specified on a per-signal basis in the **sigacts**() structure for the process. Trampoline version 0 is reserved for the legacy kernel-provided, on-stack signal trampoline. All other trampoline versions indicate a specific trampoline ABI. This ABI is coordinated with machine-dependent code in the system C library.

SIGNAL TRAMPOLINE

The signal trampoline is a special piece of code which provides support for invoking the signal handlers for a process. The trampoline is used to return from the signal handler back to the code which was executing when the signal was delivered, and is also used to invoke the handler itself on architectures where it is not feasible to have the kernel invoke the handler directly.

In traditional UNIX systems, the signal trampoline, also referred to as the "sigcode", is provided by the kernel and copied to the top of the user's stack when a new process is created or a new program image is exec'd. Starting in NetBSD 2.0, the signal trampoline is provided by the system C library. This allows for more flexibility when the signal facility is extended, makes dealing with signals easier in debuggers, such as gdb(1), and may also enhance system security by allowing the kernel to disallow execution of code on the stack.

The signal trampoline is specified on a per-signal basis. The correct trampoline is selected automatically by the C library when a signal handler is registered by a process.

Signal trampolines have a special naming convention which enables debuggers to determine the characteristics of the signal handler and its arguments. Trampoline functions are named like so:

```
__sigtramp_<flavor>_<version>
```

where:

⟨flavor⟩ The flavor of the signal handler. The following flavors are valid:

sigcontext Specifies a traditional BSD-style (deprecated) signal handler with the following signature:

```
void (*handler)(int signum,
        int code,
        struct sigcontext *scp);
```

siginfo Specifies a POSIX-style signal handler with the following signature:

```
void (*handler)(int signum,
        siginfo_t *si,
        void *uc);
```

Note: sigcontext style signal handlers are deprecated, and retained only for compatibility with older binaries.

⟨version⟩ Specifies the ABI version of the signal trampoline. The trampoline ABI is coordinated with the machine-dependent kernel **sendsig**() function. The trampoline version needs to be unique even across different trampoline flavors, in order to simplify trampoline selection in the kernel.

The following is an example if a signal trampoline name which indicates that the trampoline is used for traditional BSD-style signal handlers and implements version 1 of the signal trampoline ABI:

```
__sigtramp_sigcontext_1
```

The current signal trampoline is:

```
__sigtramp_siginfo_2
```

SEE ALSO

sigaction(2), signal(7), condvar(9)

NAME

softint, **softint_establish**, **softint_disestablish**, **softint_schedule** — machine-independent software interrupt framework

SYNOPSIS

```
#include <sys/intr.h>

void *
softint_establish(int flags, void (*func)(void *), void *arg);

void
softint_disestablish(void *cookie);

void
softint_schedule(void *cookie);
```

DESCRIPTION

The software interrupt framework is designed to provide a generic software interrupt mechanism which can be used any time a low-priority callback is needed.

It allows dynamic registration of software interrupts for loadable drivers and protocol stacks, prioritization and fair queueing of software interrupts, and allows machine-dependent optimizations to reduce cost.

Four priority levels are provided. In order of priority (lowest to highest) the levels are: clock, bio, net, serial. The names are symbolic and in isolation do not have any direct connection with a particular kind of device activity: they are only meant as a guide.

The four priority levels map directly to scheduler priority levels, and where the architecture implements 'fast' software interrupts, they also map onto interrupt priorities. The interrupt priorities are intended to be hidden from machine independent code, which should in general use thread-safe mechanisms to synchronize with software interrupts (for example: mutexes).

Software interrupts run with limited machine context. In particular, they do not possess any address space context. They should not try to operate on user space addresses, or to use virtual memory facilities other than those noted as interrupt safe. Unlike hardware interrupts, software interrupts do have thread context. They may block on synchronization objects, sleep, and resume execution at a later time.

Since software interrupts are a limited resource and run with higher priority than most other LWPs in the system, all block-and-resume activity by a software interrupt must be kept short to allow further processing at that level to continue. By extension, code running with process context must take care to ensure that any lock that may be taken from a software interrupt can not be held for more than a short period of time.

The kernel does not allow software interrupts to use facilities or perform actions that are likely to block for a significant amount of time. This means that it's not valid for a software interrupt to sleep on condition variables or to wait for resources to become available (for example, memory).

The following is a brief description of each function in the framework:

softint_establish(*flags*, *func*, *arg*)

Register a software interrupt. The *flags* value must contain one of the following constants, specifing the priority level for the soft interrupt:

SOFTINT_CLOCK, SOFTINT_BIO, SOFTINT_NET, SOFTINT_SERIAL

If the constant SOFTINT_MPSAFE is not logically ORed into *flags*, the global kernel_lock will automatically be acquired before the soft interrupt handler is called.

The constant *func* specifies the function to call when the soft interrupt is executed. The argument *arg* will be passed to this function.

softint_establish() may block in order to allocate memory. If successful, it returns a non-NULL opaque value to be used as an argument to **softint_schedule**() and/or **softint_disestablish**(). If for some reason it does not succeed, it returns NULL.

softint_disestablish(*cookie*)

Deallocate a software interrupt previously allocated by a call to **softint_establish**().

softint_schedule(*cookie*)

Schedule a software interrupt previously allocated by a call to **softint_establish**() to be executed as soon as that software interrupt is unblocked. **softint_schedule**() can safely be called multiple times before the callback routine is invoked.

Soft interrupt scheduling is CPU-local. A request to dispatch a soft interrupt will only be serviced on the same CPU where the request was made. The LWPs (light weight processes) dedicated to soft interrupt processing are bound to their home CPUs, so if a soft interrupt handler sleeps and later resumes, it will always resume on the same CPU.

On a system with multiple processors, multiple instances of the same soft interrupt handler can be in flight simultaneously (at most one per-CPU).

SEE ALSO

mutex(9), rwlock(9), spl(9)

HISTORY

The NetBSD machine-independent software interrupt framework was designed in 1997 and was implemented by one port in NetBSD 1.3. However, it did not gain wider implementation until NetBSD 1.5. Between NetBSD 4.0 and NetBSD 5.0 the framework was re-implemented in a machine-independant way to provide software interrupts with thread context.

NAME

spl, **spl0**, **splhigh**, **splvm**, **splsched**, **splsoftbio**, **splsoftclock**, **splsoftnet**, **splsoftserial**, **splx** — modify system interrupt priority level

SYNOPSIS

```
#include <sys/intr.h>

void
spl0(void);

int
splhigh(void);

int
splsched(void);

int
splvm(void);

int
splsoftbio(void);

int
splsoftclock(void);

int
splsoftserial(void);

int
splsoftnet(void);

void
splx(int s);
```

DESCRIPTION

These functions raise and lower the interrupt priority level. They are used by kernel code to block interrupts in critical sections, in order to protect data structures.

In a multi-CPU system, these functions change the interrupt priority level on the local CPU only. In general, device drivers should not make use of these interfaces. To ensure correct synchronization, device drivers should use the condvar(9), mutex(9), and rwlock(9) interfaces.

Interrupt priorities are arranged in a strict hierarchy, although sometimes levels may be equivalent (overlap). The hierarchy means that raising the IPL to any level will block interrupts at that level, and at all lower levels. The hierarchy is used to minimize data loss due to interrupts not being serviced in a timely fashion.

The levels may be divided into two groups: hard and soft. Hard interrupts are generated by hardware devices. Soft interrupts are a way of deferring hardware interrupts to do more expensive processing at a lower interrupt priority, and are explicitly scheduled by the higher-level interrupt handler. Software interrupts are further described by softint(9).

Note that hard interupt handlers do not possess process (thread) context and so it is not valid to use kernel facilities that may attempt to sleep from a hardware interrupt. For example, it is not possible to acquire a reader/writer lock from a hardware interrupt. Soft interrupt handlers possess limited process context and so may sleep briefly in order to acquire a reader/writer lock or adaptive mutex, but may not sleep for any other reason.

In order of highest to lowest priority, the priority-raising functions along with their counterpart symbolic tags are:

splhigh(), IPL_HIGH

Blocks all hard and soft interrupts, including the highest level I/O interrupts, such as interrupts from serial interfaces and the statistics clock (if any). It is also used for code that cannot tolerate any interrupts.

Code running at this level may not (in general) directly access machine independent kernel services. For example, it is illegal to call the kernel **printf**() function or to try and allocate memory. The methods of synchronization available are: spin mutexes and scheduling a soft interrupt. Generally, all code run at this level must schedule additional processing to run in a software interrupt.

Code with thread context running at this level must not use a kernel interface that may cause the current LWP to sleep, such as the `condvar` interfaces.

Interrupt handlers at this level cannot acquire the global kernel_lock and so must be coded to ensure correct synchronization on multiprocessor systems.

splsched(), IPL_SCHED

Blocks all medium priority hardware interrupts, such as interrupts from audio devices, and the clock interrupt.

Interrupt handlers running at this level endure the same restrictions as at IPL_HIGH, but may access scheduler interfaces, and so may awaken LWPs (light weight processes) using the `condvar`(9) interfaces, and may schedule callouts using the `callout`(9) interfaces.

Code with thread context running at this level may sleep via the `condvar` interfaces, and may use other kernel facilities that could cause the current LWP to sleep.

splvm(), IPL_VM

Blocks hard interrupts from 'low' priority hardware interrupts, such as interrupts from network, block I/O and tty devices.

Code running at this level endures the same restrictions as at IPL_SCHED, but may use the deprecated `malloc`(9) or endorsed `pool_cache`(9) interfaces to allocate memory.

At the time of writing, the global `kernel_lock` is automatically acquired for interrupts at this level, in order to support device drivers that do not provide their own multiprocessor synchronization. A future release of the system may allow the automatic acquisition of `kernel_lock` to be disabled for individual interrupt handlers.

splsoftserial(), IPL_SOFTSERIAL

Blocks soft interrupts at the IPL_SOFTSERIAL symbolic level.

This is the first of the software levels. Soft interrupts at this level and lower may acquire reader/writer locks or adaptive mutexes.

splsoftnet(), IPL_SOFTNET

Blocks soft interrupts at the IPL_SOFTNET symbolic level.

splsoftbio(), IPL_SOFTBIO

Blocks soft interrupts at the IPL_SOFTBIO symbolic level.

splsoftclock(), IPL_SOFTCLOCK

Blocks soft interrupts at the IPL_SOFTCLOCK symbolic level.

This is the priority at which callbacks generated by the callout(9) facility runs.

One function lowers the system priority level:

spl0(), IPL_NONE

Unblocks all interrupts. This should rarely be used directly; **splx**() should be used instead.

The **splx**() function restores the system priority level to the one encoded in *s*, which must be a value previously returned by one of the other **spl** functions.

SEE ALSO

condvar(9), mutex(9), rwlock(9)

HISTORY

In 4.4BSD, **splnet**() was used to block network software interrupts. Most device drivers used **splimp**() to block hardware interrupts. To avoid unnecessarily blocking other interrupts, in NetBSD 1.1 a new function was added that blocks only network hardware interrupts. For consistency with other **spl** functions, the old **splnet**() function was renamed to **splsoftnet**(), and the new function was named **splnet**().

Originally, **splsoftclock**() lowered the system priority level. During the NetBSD 1.5 development cycle, **spllowersoftclock**() was introduced and the semantics of **splsoftclock**() were changed.

The **splimp**() call was removed from the kernel between NetBSD 1.5 and NetBSD 1.6. The function of **splimp**() was replaced by **splvm**() and code which abused the semantics of **splimp**() was changed to not mix interrupt priority levels.

Between NetBSD 4.0 and NetBSD 5.0, the hardware levels were reduced in number and a strict hierarchy defined.

NAME

splraiseipl — raise the system priority level

SYNOPSIS

```
#include <sys/param.h>

int
splraiseipl(ipl_cookie_t icookie);
```

DESCRIPTION

splraiseipl() raises the system priority level to the level specified by *icookie*. *icookie* should be a value returned by **makeiplcookie**().

In general, device drivers should not make use of this interface. To ensure correct synchronization, device drivers should use the condvar(9), mutex(9), and rwlock(9) interfaces.

See the spl(9) manual page for a description of interrupt priority levels.

RETURN VALUES

splraiseipl() returns saved priority level which can be used for **splx**().

EXAMPLES

The following two lines are functional equivalents.

```
s = splraiseipl(makeiplcookie(IPL_VM));

s = splvm();
```

Because **makeiplcookie**() can be slow and is not expected to be used in a perfomance critical path, it's better to do it beforehand.

```
initialization_code(ipl_t ipl)
{

        ourcookie = makeiplcookie(ipl);
}

performance_critical_code()
{
        int s;

        s = splraiseipl(ourcookie);
        do_something();
        splx(s);
}
```

SEE ALSO

condvar(9), makeiplcookie(9), mutex(9), rwlock(9), spl(9)

NAME

store, **subyte**, **suibyte**, **susword**, **suswintr**, **suword**, **suiword** — store data to user-space

SYNOPSIS

```
#include <sys/types.h>
#include <sys/systm.h>

int
subyte(void *base, int c);

int
susword(void *base, short c);

int
suswintr(void *base, short c);

int
suword(void *base, long c);
```

DESCRIPTION

The **store** functions are designed to copy small amounts of data to the user-space of the currently running process.

The **store** routines provide the following functionality:

subyte() Stores a byte of data to the user-space address *base*.

susword() Stores a short word of data to the user-space address *base*.

suswintr() Stores a short word of data to the user-space address *base*. This function is safe to call during an interrupt context.

suword() Stores a word of data to the user-space address *base*.

RETURN VALUES

The **store** functions return 0 on success or -1 on failure.

SEE ALSO

copy(9), fetch(9)

NAME

suspendsched — suspend the scheduler

SYNOPSIS

```
#include <sys/proc.h>
#include <sys/sched.h>

void
suspendsched(void);
```

DESCRIPTION

The **suspendsched**() function suspends the operation of the scheduler by stopping all non-system processes which are on the run queue or the sleep queue.

The **suspendsched**() function must not be called with the scheduler lock held.

SEE ALSO

scheduler(9)

NAME

sysctl — system variable control interfaces

SYNOPSIS

```
#include <sys/param.h>
#include <sys/sysctl.h>
```

Primary external interfaces:

```
void
sysctl_init(void);

int
sysctl_lock(struct lwp *l, void *oldp, size_t savelen);

int
sysctl_dispatch(const int *name, u_int namelen, void *oldp, size_t *oldlenp,
    const void *newp, size_t newlen, const int *oname, struct lwp *l,
    const struct sysctlnode *rnode);

void
sysctl_unlock(struct lwp *l);

int
sysctl_createv(struct sysctllog **log, int cflags,
    const struct sysctlnode **rnode, const struct sysctlnode **cnode,
    int flags, int type, const char *namep, const char *desc,
    sysctlfn func, u_quad_t qv, void *newp, size_t newlen, ...);

int
sysctl_destroyv(struct sysctlnode *rnode, ...);

void
sysctl_free(struct sysctlnode *rnode);

void
sysctl_teardown(struct sysctllog **);

int
old_sysctl(int *name, u_int namelen, void *oldp, size_t *oldlenp,
    void *newp, size_t newlen, struct lwp *l);
```

Core internal functions:

```
int
sysctl_locate(struct lwp *l, const int *name, u_int namelen,
    const struct sysctlnode **rnode, int *nip);

int
sysctl_lookup(const int *name, u_int namelen, void *oldp, size_t *oldlenp,
    const void *newp, size_t newlen, const int *oname, struct lwp *l,
    const struct sysctlnode *rnode);

int
sysctl_create(const int *name, u_int namelen, void *oldp, size_t *oldlenp,
    const void *newp, size_t newlen, const int *oname, struct lwp *l,
    const struct sysctlnode *rnode);
```

int
sysctl_destroy(*const int *name*, *u_int namelen*, *void *oldp*, *size_t *oldlenp*,
*const void *newp*, *size_t newlen*, *const int *oname*, *struct lwp *l*,
*const struct sysctlnode *rnode*);

int
sysctl_query(*const int *name*, *u_int namelen*, *void *oldp*, *size_t *oldlenp*,
*const void *newp*, *size_t newlen*, *const int *oname*, *struct lwp *l*,
*const struct sysctlnode *rnode*);

Simple "helper" functions:

int
sysctl_needfunc(*const int *name*, *u_int namelen*, *void *oldp*, *size_t *oldlenp*,
*const void *newp*, *size_t newlen*, *const int *oname*, *struct lwp *l*,
*const struct sysctlnode *rnode*);

int
sysctl_notavail(*const int *name*, *u_int namelen*, *void *oldp*, *size_t *oldlenp*,
*const void *newp*, *size_t newlen*, *const int *oname*, *struct lwp *l*,
*const struct sysctlnode *rnode*);

int
sysctl_null(*const int *name*, *u_int namelen*, *void *oldp*, *size_t *oldlenp*,
*const void *newp*, *size_t newlen*, *const int *oname*, *struct lwp *l*,
*const struct sysctlnode *rnode*);

DESCRIPTION

The SYSCTL subsystem instruments a number of kernel tunables and other data structures via a simple MIB-like interface, primarily for consumption by userland programs, but also for use internally by the kernel.

LOCKING

All operations on the SYSCTL tree must be protected by acquiring the main SYSCTL lock. The only functions that can be called when the lock is not held are **sysctl_lock**(), **sysctl_createv**(), **sysctl_destroyv**(), and **old_sysctl**(). All other functions require the tree to be locked. This is to prevent other users of the tree from moving nodes around during an add operation, or from destroying nodes or subtrees that are actively being used. The lock is acquired by calling **sysctl_lock**() with a pointer to the process's lwp *l* (`NULL` may be passed to all functions as the lwp pointer if no lwp is appropriate, though any changes made via **sysctl_create**(), **sysctl_destroy**(), **sysctl_lookup**(), or by any helper function will be done with effective superuser privileges). The *oldp* and *savelen* arguments are a pointer to and the size of the memory region the caller will be using to collect data from SYSCTL. These may also be `NULL` and 0, respectively.

The memory region will be locked via **uvm_vslock**() if it is a region in userspace. The address and size of the region are recorded so that when the SYSCTL lock is to be released via **sysctl_unlock**(), only the lwp pointer *l* is required.

LOOKUPS

Once the lock has been acquired, it is typical to call **sysctl_dispatch**() to handle the request. **sysctl_dispatch**() will examine the contents of *name*, an array of integers at least *namelen* long, which is to be located in kernel space, in order to determine which function to call to handle the specific request.

sysctl_dispatch() uses the following algorithm to determine the function to call:

- Scan the tree using **sysctl_locate**()
- If the node returned has a "helper" function, call it
- If the requested node was found but has no function, call **sysctl_lookup**()
- If the node was not found and *name* specifies one of **sysctl_query**(), **sysctl_create**(), or **sysctl_destroy**(), call the appropriate function
- If none of these options applies and no other error was yet recorded, return EOPNOTSUPP

The *oldp* and *oldlenp* arguments to **sysctl_dispatch**(), as with all the other core functions, describe an area into which the current or requested value may be copied. *oldp* may or may not be a pointer into userspace (as dictated by whether *l* is NULL or not). *oldlenp* is a non-NULL pointer to a size_t. *newp* and *newlen* describe an area where the new value for the request may be found; *newp* may also be a pointer into userspace. The *oname* argument is a non-NULL pointer to the base of the request currently being processed. By simple arithmetic on *name*, *namelen*, and *oname*, one can easily determine the entire original request and *namelen* values, if needed. The *rnode* value, as passed to **sysctl_dispatch**() represents the root of the tree into which the current request is to be dispatched. If NULL, the main tree will be used.

sysctl_locate() scans a tree for the node most specific to a request. If the pointer referenced by *rnode* is not NULL, the tree indicated is searched, otherwise the main tree will be used. The address of the most relevant node will be returned via *rnode* and the number of MIB entries consumed will be returned via *nip*, if it is not NULL.

The **sysctl_lookup**() function takes the same arguments as **sysctl_dispatch**() with the caveat that the value for *namelen* must be zero in order to indicate that the node referenced by the *rnode* argument is the one to which the lookup is being applied.

CREATION AND DESTRUCTION OF NODES

New nodes are created and destroyed by the **sysctl_create**() and **sysctl_destroy**() functions. These functions take the same arguments as **sysctl_dispatch**() with the additional requirement that the *namelen* argument must be 1 and the *name* argument must point to an integer valued either CTL_CREATE or CTL_CREATESYM when creating a new node, or CTL_DESTROY when destroying a node. The *newp* and *newlen* arguments should point to a copy of the node to be created or destroyed. If the create or destroy operation was successful, a copy of the node created or destroyed will be placed in the space indicated by *oldp* and *oldlenp*. If the create operation fails because of a conflict with an existing node, a copy of that node will be returned instead.

In order to facilitate the creation and destruction of nodes from a given tree by kernel subsystems, the functions **sysctl_createv**() and **sysctl_destroyv**() are provided. These functions take care of the overhead of filling in the contents of the create or destroy request, dealing with locking, locating the appropriate parent node, etc.

The arguments to **sysctl_createv**() are used to construct the new node. If the *log* argument is not NULL, a sysctllog structure will be allocated and the pointer referenced will be changed to address it. The same log may be used for any number of nodes, provided they are all inserted into the same tree. This allows for a series of nodes to be created and later removed from the tree in a single transaction (via **sysctl_teardown**()) without the need for any record keeping on the caller's part. The *cflags* argument is currently unused and must be zero. The *rnode* argument must either be NULL or a valid pointer to a reference to the root of the tree into which the new node must be placed. If it is NULL, the main tree will be used. It is illegal for *rnode* to refer to a NULL pointer. If the *cnode* argument is not NULL, on return it will be adjusted to point to the address of the new node.

The *flags* and *type* arguments are combined into the *sysctl_flags* field, and the current value for SYSCTL_VERSION is added in. Note: the CTLFLAG_PERMANENT flag can only be set from SYSCTL setup routines (see **SETUP FUNCTIONS**) as called by **sysctl_init**(). The *namep* argument is copied into the *sysctl_name* field and must be less than SYSCTL_NAMELEN characters in length. The string indicated by *desc* will be copied if the CTLFLAG_OWNDESC flag is set, and will be used as the node's description. Note: if **sysctl_destroyv**() attempts to delete a node that does not own its own description (and is not marked as permanent), but the deletion fails, the description will be copied and **sysctl_destroyv**() will set the CTLFLAG_OWNDESC flag.

The *func* argument is the name of a "helper" function (see **HELPER FUNCTIONS AND MACROS**). If the CTLFLAG_IMMEDIATE flag is set, the *qv* argument will be interpreted as the initial value for the new "int" or "quad" node. This flag does not apply to any other type of node. The *newp* and *newlen* arguments describe the data external to SYSCTL that is to be instrumented. One of *func*, *qv* and the CTLFLAG_IMMEDIATE flag, or *newp* and *newlen* must be given for nodes that instrument data, otherwise an error is returned.

The remaining arguments are a list of integers specifying the path through the MIB to the node being created. The list must be terminated by the CTL_EOL value. The penultimate value in the list may be CTL_CREATE if a dynamic MIB entry is to be made for this node. **sysctl_createv**() specifically does not support CTL_CREATESYM, since setup routines are expected to be able to use the in-kernel ksyms(4) interface to discover the location of the data to be instrumented. If the node to be created matches a node that already exists, a return code of 0 is given, indicating success.

When using **sysctl_destroyv**() to destroy a given node, the *rnode* argument, if not NULL, is taken to be the root of the tree from which the node is to be destroyed, otherwise the main tree is used. The rest of the arguments are a list of integers specifying the path through the MIB to the node being destroyed. If the node being destroyed does not exist, a successful return code is given. Nodes marked with the CTLFLAG_PERMANENT flag cannot be destroyed.

HELPER FUNCTIONS AND MACROS

Helper functions are invoked with the same common argument set as **sysctl_dispatch**() except that the *rnode* argument will never be NULL. It will be set to point to the node that corresponds most closely to the current request. Helpers are forbidden from modifying the node they are passed; they should instead copy the structure if changes are required in order to effect access control or other checks. The "helper" prototype and function that needs to ensure that a newly assigned value is within a certain range (presuming external data) would look like the following:

```
static int sysctl_helper(SYSCTLFN_PROTO);

static int
sysctl_helper(SYSCTLFN_ARGS)
{
        struct sysctlnode node;
        int t, error;

        node = *rnode;
        node.sysctl_data = &t;
        error = sysctl_lookup(SYSCTLFN_CALL(&node));
        if (error || newp == NULL)
                return (error);

        if (t < 0 || t > 20)
                return (EINVAL);
```

```
        *(int*)rnode->sysctl_data = t;
        return (0);
}
```

The use of the SYSCTLFN_PROTO, SYSCTLFN_ARGS, and SYSCTLFN_CALL macros ensure that all arguments are passed properly. The single argument to the SYSCTLFN_CALL macro is the pointer to the node being examined.

Three basic helper functions are available for use. **sysctl_needfunc**() will emit a warning to the system console whenever it is invoked and provides a simplistic read-only interface to the given node. **sysctl_notavail**() will forward "queries" to **sysctl_query**() so that subtrees can be discovered, but will return EOPNOTSUPP for any other condition. **sysctl_null**() specifically ignores any arguments given, sets the value indicated by *oldlenp* to zero, and returns success.

SETUP FUNCTIONS

Though nodes can be added to the SYSCTL tree at any time, in order to add nodes during the kernel bootstrap phase, a proper "setup" function must be used. Setup functions are declared using the SYSCTL_SETUP macro, which takes the name of the function and a short string description of the function as arguments. The address of the function is added to a list of functions that **sysctl_init**() traverses during initialization.

Setup functions do not have to add nodes to the main tree, but can set up their own trees for emulation or other purposes. Emulations that require use of a main tree but with some nodes changed to suit their own purposes can arrange to overlay a sparse private tree onto their main tree by making the *e_sysctlovly* member of their struct emul definition point to the overlaid tree.

Setup functions should take care to create all nodes from the root down to the subtree they are creating, since the order in which setup functions are called is arbitrary (the order in which setup functions are called is only determined by the ordering of the object files as passed to the linker when the kernel is built).

MISCELLANEOUS FUNCTIONS

sysctl_init() is called early in the kernel bootstrap process. It initializes the SYSCTL lock, calls all the registered setup functions, and marks the tree as permanent.

sysctl_free() will unconditionally delete any and all nodes below the given node. Its intended use is for the deletion of entire trees, not subtrees. If a subtree is to be removed, **sysctl_destroy**() or **sysctl_destroyv**() should be used to ensure that nodes not owned by the sub-system being deactivated are not mistakenly destroyed. The SYSCTL lock must be held when calling this function.

sysctl_teardown() unwinds a sysctllog and deletes the nodes in the opposite order in which they were created.

old_sysctl() provides an interface similar to the old SYSCTL implementation, with the exception that access checks on a per-node basis are performed if the *l* argument is non-NULL. If called with a NULL argument, the values for *newp* and *oldp* are interpreted as kernel addresses, and access is performed as for the superuser.

NOTES

It is expected that nodes will be added to (or removed from) the tree during the following stages of a machine's lifetime:

- initialization -- when the kernel is booting
- autoconfiguration -- when devices are being probed at boot time

- "plug and play" device attachment -- when a PC-Card, USB, or other device is plugged in or attached
- LKM initialization -- when an LKM is being loaded
- "run-time" -- when a process creates a node via the `sysctl`(3) interface

Nodes marked with `CTLFLAG_PERMANENT` can only be added to a tree during the first or initialization phase, and can never be removed. The initialization phase terminates when the main tree's root is marked with the `CTLFLAG_PERMANENT` flag. Once the main tree is marked in this manner, no nodes can be added to any tree that is marked with `CTLFLAG_READONLY` at its root, and no nodes can be added at all if the main tree's root is so marked.

Nodes added by device drivers, LKMs, and at device insertion time can be added to (and removed from) "read-only" parent nodes.

Nodes created by processes can only be added to "writable" parent nodes. See `sysctl`(3) for a description of the flags that are allowed to be used by when creating nodes.

SEE ALSO

`sysctl`(3)

HISTORY

The dynamic SYSCTL implementation first appeared in NetBSD 2.0.

AUTHORS

Andrew Brown ⟨atatat@NetBSD.org⟩ designed and implemented the dynamic SYSCTL implementation.

NAME

sysmon_envsys — kernel part of the envsys 2 framework

SYNOPSIS

```
#include <dev/sysmon/sysmonvar.h>

struct sysmon_envsys *
sysmon_envsys_create(void);

void
sysmon_envsys_destroy(struct sysmon_envsys *);

int
sysmon_envsys_register(struct sysmon_envsys *);

void
sysmon_envsys_unregister(struct sysmon_envsys *);

int
sysmon_envsys_sensor_attach(struct sysmon_envsys *, envsys_data_t *);

int
sysmon_envsys_sensor_detach(struct sysmon_envsys *, envsys_data_t *);
```

DESCRIPTION

sysmon_envsys is the kernel part of the envsys(4) framework. With this framework you are able to register and unregister a **sysmon_envsys** device, attach or detach sensors into a device and enable or disable automatic monitoring for some sensors without any user interactivity, among other things.

HOW TO USE THE FRAMEWORK

To register a new driver to the **sysmon_envsys** framework, a **sysmon_envsys** object must be allocated and initialized; the **sysmon_envsys_create**() function is used for this. This returns a zero'ed pointer to a sysmon_envsys structure and takes care of initialization of some private features.

Once we have the object we could start initializing sensors (see the *SENSOR DETAILS* section for more information) and attaching them to the device, this is acomplished by the **sysmon_envsys_sensor_attach**() function. This function attachs the envsys_data_t (sensor) specified as second argument into the sysmon_envsys object specified in the first argument.

Finally when the sensors are already attached, the device needs to set some required (and optional) members of the sysmon_envsys struct before calling the **sysmon_envsys_register**() function to register the device.

If there's some error before registering the device, the **sysmon_envsys_destroy**() function must be used to detach the sensors previously attached and free the sysmon_envsys object allocated by the **sysmon_envsys_create**() function.

The *sysmon_envsys* structure is defined as follow (only the public members are shown):

```
struct sysmon_envsys {
        const char      *sme_name;
        int             sme_flags;
        int             sme_class;
        uint64_t        sme_events_timeout;
        void            *sme_cookie;
        void (*sme_refresh)(struct sysmon_envsys *, envsys_data_t *);
};
```

The members have the following meaning:

sme_class This specifies the class of the sysmon envsys device. See the **DEVICE CLASSES** section for more information (OPTIONAL).

sme_name The name that will be used in the driver (REQUIRED).

sme_flags Additional flags for the **sysmon_envsys** device. Currently supporting *SME_DISABLE_REFRESH*. If enabled, the *sme_refresh* function callback won't be used to refresh sensors data and the driver will use its own method. Hence *sme_cookie* won't be necessary either (OPTIONAL).

sme_events_timeout This is used to specify the default timeout value that will be used to check for critical events if any monitoring flag was set. The value is used as seconds (OPTIONAL).

If the driver wants to refresh sensors data via the **sysmon_envsys** framework, the following members must be specified:

sme_cookie Pointer to the device struct (also called "softc"). This may be used in the **sme_refresh** function callback.

sme_refresh Pointer to a function that will be used to refresh sensor data in the device. This can be used to set the state and other properties of the sensor depending of the returned data by the driver. *NOTE*: *You don't have to refresh all sensors, only the sensor specified by the* **edata->sensor** *index.*

Note that it's not necessary to refresh the sensors data before the driver is registered, only do it if you need the data in your driver to check for a specific condition.

The timeout value for the monitoring events on a device may be changed via the *ENVSYS_SETDICTIONARY* `ioctl`(2) or the `envstat`(8) command.

To unregister a driver previously registered with the **sysmon_envsys** framework, the **sysmon_envsys_unregister**() function must be used. If there were monitoring events registered for the driver, they all will be destroyed before the device is unregistered and its sensors will be detached; finally the *sysmon_envsys* object will be freed, so there's no need to call **sysmon_envsys_destroy**() if we are going to unregister a device.

DEVICE CLASSES

The *sme_class* member of the *sysmon_envsys* structure is an optional flag that specifies the class of the sysmon envsys device. Currently there are two classes:

SME_CLASS_ACADAPTER

This class is for devices that want to act as an *AC adapter*. The device writer must ensure that at least there is a sensor with *units* of **ENVSYS_INDICATOR**. This will be used to report its current state (on/off).

SME_CLASS_BATTERY

This class is for devices that want to act as an *Battery*. The device writer must ensure that at least there are two sensors with units of **ENVSYS_BATTERY_CAPACITY** and **ENVSYS_BATTERY_CHARGE**.

These two sensors are used to ensure that the battery device won't never send a *low-power* event to the `powerd`(8) daemon (if running) when all battery devices are in a critical state. The critical state means that a battery is not currently charging and its charge state is low or critical. When the

low-power condition is met, an event is sent to the powerd(8) daemon (if running) and will shutdown the system gracefully via the */etc/powerd/scripts/sensor_battery* script.

If powerd(8) is not running, the system will be powered off via the cpu_reboot(9) call with the *RB_POWERDOWN* flag.

NOTE: If a *SME_CLASS_ACADAPTER* or *SME_CLASS_BATTERY* class don't have the sensors required, the *low-power* event will never be sent, and the graceful shutdown won't be possible.

SENSOR DETAILS

Each sensor uses a **envsys_data_t** structure, it's defined as follow (only the public members are shown);

```
typedef struct envsys_data {
        uint32_t        units;
        uint32_t        state;
        uint32_t        flags;
        uint32_t        rpms;
        int32_t         rfact;
        int32_t         value_cur;
        int32_t         value_max;
        int32_t         value_min;
        int32_t         value_avg;
        bool            monitor;
        char            desc[ENVSYS_DESCLEN];
} envsys_data_t;
```

The members for the **envsys_data_t** structure have the following meaning:

units Used to set the units type.

state Used to set the current state.

flags Used to set additional flags.

rpms Used to set the nominal RPM value for **fan** sensors.

rfact Used to set the rfact value for **voltage** sensors.

value_cur Used to set the current value.

value_max Used to set the maximum value.

value_min Used to set the minimum value.

value_avg Used to set the average value.

monitor Used to enable automatic sensor monitoring (by default it's disabled). The automatic sensor monitoring will check if a condition is met periodically and will send an event to the powerd(8) daemon (if running). The monitoring event will be registered when this flag is *true* and one or more of the *ENVSYS_FMONFOO* flags were set in the *flags* member.

desc Used to set the description string. *NOTE that the description string must be unique in a device, and sensors with duplicate or empty description will simply be ignored.*

Users of this framework must take care about the following points:

- The *desc* member needs to have a valid description, unique in a device and non empty to be valid.

- The *units* type must be valid. The following units are defined:

ENVSYS_STEMP	For temperature sensors.
ENVSYS_SFANRPM	For fan sensors.
ENVSYS_SVOLTS_AC	For AC Voltage.
ENVSYS_SVOLTS_DC	For DC Voltage.
ENVSYS_SOHMS	For Ohms.
ENVSYS_SWATTS	For Watts.
ENVSYS_SAMPS	For Ampere.
ENVSYS_SWATTHOUR	For Watts hour.
ENVSYS_SAMPHOUR	For Ampere hour.
ENVSYS_INDICATOR	For sensors that only want a boolean type.
ENVSYS_INTEGER	For sensors that only want an integer type.
ENVSYS_DRIVE	For drive sensors.
ENVSYS_BATTERY_CAPACITY	For Battery device classes. This sensor unit uses the ENVSYS_BATTERY_CAPACITY_* values in *value_cur* to report its current capacity to userland. Mandatory if *sme_class* is set to *SME_CLASS_BATTERY*.
ENVSYS_BATTERY_CHARGE	For Battery device classes. This sensor is equivalent to the Indicator type, it's a boolean. Use it to specify in what state is the Battery state: **true** if the battery is currently charging or **false** otherwise. Mandatory if *sme_class* is set to *SME_CLASS_BATTERY*.

- When initializing or refreshing the sensor, the *state* member should be set to a known state (otherwise it will be in unknown state). Possible values:

ENVSYS_SVALID	Sets the sensor to a valid state.
ENVSYS_SINVALID	Sets the sensor to an invalid state.
ENVSYS_SCRITICAL	Sets the sensor to a critical state.
ENVSYS_SCRITUNDER	Sets the sensor to a critical under state.
ENVSYS_SCRITOVER	Sets the sensor to a critical over state.
ENVSYS_SWARNUNDER	Sets the sensor to a warning under state.
ENVSYS_SWARNOVER	Sets the sensor to a warning over state.

- The *flags* member accepts one or more of the following flags:

ENVSYS_FCHANGERFACT	Marks the sensor with ability to change the *rfact* value on the fly (in voltage sensors). The *rfact* member must be used in the correct place of the code that retrieves and converts the value of the sensor.
ENVSYS_FPERCENT	This uses the *value_cur* and *value_max* members to make a percentage. Both values must be enabled and have data.
ENVSYS_FVALID_MAX	Marks the *value_max* value as valid.
ENVSYS_FVALID_MIN	Marks the *value_min* value as valid.
ENVSYS_FVALID_AVG	Marks the *value_avg* value as valid.
ENVSYS_FMONCRITICAL	Enables and registers a new event to monitor a critical state.
ENVSYS_FMONCRITUNDER	Enables and registers a new event to monitor a critical under state.
ENVSYS_FMONCRITOVER	Enables and registers a new event to monitor a critical over state.

ENVSYS_FMONWARNUNDER
: Enables and registers a new event to monitor a warning under state.

ENVSYS_FMONWARNOVER
: Enables and registers a new event to monitor a warning over state.

ENVSYS_FMONSTCHANGED
: Enables and registers a new event to monitor Battery capacity or drive state sensors. It won't be effective if the *units* member is not set to *ENVSYS_DRIVE* or *ENVSYS_BATTERY_CAPACITY*.

ENVSYS_FMONNOTSUPP
: Disallows to set a critical limit via the *ENVSYS_SETDICTIONARY* ioctl(2). This flag has not any effect for monitoring flags set in the driver and it's only meant to disable setting critical limits from userland.

If the driver has to use any of the value_max, value_min or value_avg members, they should be marked as valid with the appropiate flag.

- If *units* is set to *ENVSYS_DRIVE*, there are some predefined states that must be set (only one) to the *value_cur* member:

ENVSYS_DRIVE_EMPTY	Drive state is unknown.
ENVSYS_DRIVE_READY	Drive is ready.
ENVSYS_DRIVE_POWERUP	Drive is powering up.
ENVSYS_DRIVE_ONLINE	Drive is online.
ENVSYS_DRIVE_OFFLINE	Drive is offline.
ENVSYS_DRIVE_IDLE	Drive is idle.
ENVSYS_DRIVE_ACTIVE	Drive is active.
ENVSYS_DRIVE_BUILD	Drive is building.
ENVSYS_DRIVE_REBUILD	Drive is rebuilding.
ENVSYS_DRIVE_POWERDOWN	Drive is powering down.
ENVSYS_DRIVE_FAIL	Drive has failed.
ENVSYS_DRIVE_PFAIL	Drive has been degraded.
ENVSYS_DRIVE_MIGRATING	Drive is migrating.
ENVSYS_DRIVE_CHECK	Drive is checking its state.

- If *units* is set to *ENVSYS_BATTERY_CAPACITY*, there are some predefined capacity states that must be set (only one) to the *value_cur* member:

ENVSYS_BATTERY_CAPACITY_NORMAL	Battery charge is in normal capacity.
ENVSYS_BATTERY_CAPACITY_CRITICAL	Battery charge is in critical capacity.
ENVSYS_BATTERY_CAPACITY_LOW	Battery charge is in low capacity.
ENVSYS_BATTERY_CAPACITY_WARNING	Battery charge is in warning capacity.

- The envsys(4) framework expects to have the values converted to a unit that can be converted to another one easily. That means the user should convert the value returned by the driver to the appropiate unit. For example voltage sensors to **mV**, temperature sensors to **uK**, Watts to **mW**, Ampere to **mA**, etc.

  The following types shouldn't need any conversion: *ENVSYS_BATTERY_CAPACITY*, *ENVSYS_BATTERY_CHARGE*, *ENVSYS_INDICATOR*, *ENVSYS_INTEGER* and *ENVSYS_DRIVE*.

  PLEASE NOTE THAT YOU MUST AVOID USING FLOATING POINT OPERATIONS IN KERNEL WHEN CONVERTING THE DATA RETURNED BY THE DRIVER TO THE APPROPIATE UNIT, IT'S NOT ALLOWED.

HOW TO ENABLE AUTOMATIC MONITORING IN SENSORS

The following example illustrates how to enable automatic monitoring in a virtual driver for a *critical* state in the first sensor *(sc_sensor[0]):*

```
int
mydriver_initialize_sensors(struct mysoftc *sc)
{
        ...
        /* sensor is initialized with a valid state */
        sc->sc_sensor[0].state = ENVSYS_SVALID;

        /*
         * the monitor member must be true to enable
         * automatic monitoring.
         */
        sc->sc_sensor[0].monitor = true;

        /* and now we specify the type of the monitoring event */
        sc->sc_sensor[0].flags |= ENVSYS_FMONCRITICAL;
        ...
}

int
mydriver_refresh(struct sysmon_envsys *sme, envsys_data_t *edata)
{
        struct mysoftc *sc = sme->sme_cookie;

        /* we get current data from the driver */
        edata->value_cur = sc->sc_getdata();

        /*
         * if value is too high, mark the sensor in
         * critical state.
         */
        if (edata->value_cur > MYDRIVER_SENSOR0_HIWAT) {
                edata->state = ENVSYS_SCRITICAL;
                /* a critical event will be sent now automatically */
        } else {
                /*
                 * if value is within the limits, and we came from
                 * a critical state make sure to change sensor's state
                 * to valid.
                 */
                edata->state = ENVSYS_SVALID;
        }
        ...
}
```

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing the **envsys 2** framework can be found. All pathnames are relative to `/usr/src`.

The **envsys 2** framework is implemented within the files:

`sys/dev/sysmon/sysmon_envsys.c`

`sys/dev/sysmon/sysmon_envsys_events.c`

`sys/dev/sysmon/sysmon_envsys_tables.c`

`sys/dev/sysmon/sysmon_envsys_util.c`

There's an example LKM driver that shows how the framework works in: `sys/lkm/misc/envsys2/lkminit_envsys2.c`.

SEE ALSO

`envsys`(4), `envstat`(8)

HISTORY

The first *envsys* framework first appeared in NetBSD 1.5. The *envsys 2* framework first appeared in NetBSD 5.0.

AUTHORS

The (current) *envsys 2* framework was implemented by Juan Romero Pardines. Additional input on the design was provided by many NetBSD developers around the world.

The first *envsys* framework was implemented by Jason R. Thorpe, Tim Rightnour and Bill Squier.

NAME

TC, **tc_intr_establish**, **tc_intr_disestablish**, **tc_intr_evcnt**. **tc_mb**, **tc_wmb**, **tc_syncbus**, **tc_badaddr**, **TC_DENSE_TO_SPARSE**, **TC_PHYS_TO_UNCACHED** — TURBOchannel bus

SYNOPSIS

```
#include <machine/bus.h>
#include <dev/tc/tcvar.h>
#include <dev/tc/tcdevs.h>

void
tc_intr_establish(struct device *dev, void *cookie, int level,
    int (*handler)(void *), void *arg);

void
tc_intr_disestablish(struct device *dev, void *cookie);

const struct evcnt *
tc_intr_evcnt(struct device *dev, void *cookie);

void
tc_mb();

void
tc_wmb();

void
tc_syncbus();

int
tc_badaddr(tc_addr_t tcaddr);

tc_addr_t
TC_DENSE_TO_SPARSE(tc_addr_t addr);

tc_addr_t
TC_PHYS_TO_UNCACHED(tc_addr_t addr);
```

DESCRIPTION

The **TC** device provides support for the DEC TURBOchannel bus found on all DEC TURBOchannel machines with MIPS (DECstation 5000 series, excluding the 5000/200) and Alpha (3000-series) systems. TURBOchannel is a 32-bit wide synchronous DMA-capable bus, running at 25 MHz on higher-end machines and at 12.5 MHz on lower-end machines.

DATA TYPES

Drivers for devices attached to the TURBOchannel bus will make use of the following data types:

struct tc_attach_args

A structure use to inform the driver of TURBOchannel bus properties. It contains the following members:

```
bus_space_tag_t        ta_memt;
bus_dma_tag_t  ta_dmat;
char           ta_modname[TC_ROM_LLEN+1];
u_int          ta_slot;
tc_offset_t    ta_offset;
tc_addr_t      ta_addr;
```

```
void            *ta_cookie;
u_int           ta_busspeed;
```

The *ta_busspeed* member specifies the TURBOchannel bus speed and is useful for time-related functions. Values values are *TC_SPEED_12_5_MHZ* for the 12.5 MHz bus and *TC_SPEED_25_MHZ* for the 50 MHz bus.

FUNCTIONS

tc_intr_establish(*dev*, *cookie*, *level*, *handler*, *arg*)
Establish an interrupt handler with device *dev* for the interrupt described completely by *cookie*, the value passed to the driver in the *ta_cookie* member of the *tc_attach_args* structure. The priority of the interrupt is specified by *level*. When the interrupt occurs the function *handler* is called with argument *arg*.

tc_intr_disestablish(*dev*, *cookie*)
Dis-establish the interrupt handler with device *dev* for the interrupt described completely *cookie*.

tc_intr_evcnt(*dev*, *cookie*)
Do interrupt event counting with device *dev* for the event described completely by *cookie*.

tc_mb() A read/write memory barrier. Any CPU-to-memory reads/writes before the barrier must complete before any CPU-to-memory reads/writes after it.

tc_wmb()
A write memory barrier. Any CPU-to-memory writes before the barrier must complete before any CPU-to-memory writes after it.

tc_syncbus()
Synchronise writes on the TURBOchannel bus by ensuring CPU writes are propagated across the TURBOchannel bus.

tc_badaddr(*tcaddr*)
Returns non-zero if the given address *tcaddr* is invalid.

TC_DENSE_TO_SPARSE(*addr*)
Convert the given physical address *addr* in TURBOchannel dense space to the corresponding address in TURBOchannel sparse space.

TC_PHYS_TO_UNCACHED(*addr*)
Convert the given system memory physical address *addr* to the physical address of the corresponding region that is not cached.

AUTOCONFIGURATION

The TURBOchannel bus is a direct-connection bus. During autoconfiguration, the parent specifies the name of the found TURBOchannel module into the *ta_modname* member of the *tc_attach_args* structure. Drivers should match on this name.

DMA SUPPORT

The TURBOchannel bus supports 32-bit, bidirectional DMA transfers. Support is provided by the standard bus_dma(9) interface.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent TURBOchannel subsystem can be found. All pathnames are relative to /usr/src.

The TURBOchannel subsystem itself is implemented within the file `sys/dev/tc/tc_subr.c`. Machine-dependent portions can be found in `sys/arch/<arch>/tc/tcbus.c`.

SEE ALSO

`tc`(4), `autoconf`(9), `bus_dma`(9), `bus_space`(9), `driver`(9)

NAME

tcp_congctl — TCP congestion control API

SYNOPSIS

```
#include <netinet/tcp_congctl.h>

int
tcp_congctl_register(const char *, struct tcp_congctl *);

int
tcp_congctl_unregister(const char *);
```

DESCRIPTION

The **tcp_congctrl** API is used to add or remove TCP congestion control algorithms on-the-fly and to modularize them. It includes basically two functions:

tcp_congctl_register(*const char **, *struct tcp_congctl **)

Registers a new congestion control algorithm. The *struct tcp_congctl* argument must contain a list of callbacks like the following:

```
struct tcp_congctl {
        int  (*fast_retransmit)(struct tcpcb *,
            struct tcphdr *);
        void (*slow_retransmit)(struct tcpcb *);
        void (*fast_retransmit_newack)(struct tcpcb *,
            struct tcphdr *);
        void (*newack)(struct tcpcb *,
            struct tcphdr *);
        void (*cong_exp)(struct tcpcb *);
};
```

tcp_congctl_unregister(*const char **)

If found, unregister the selected TCP congestion control algorithm.

RETURN VALUES

tcp_congctl_register() and **tcp_congctl_unregister**() both return `0` when there is no error. If the name is already registered, **tcp_congctl_register**() will return `EEXIST`. **tcp_congctl_unregister**() can return `ENOENT` if there is no congestion control algorithm by that name and can return `EBUSY` if the matched algorithm is being used by userspace applications.

FILES

Implementation is in `sys/netinet/tcp_congctl.c` and the interface is in `sys/netinet/tcp_congctl.h`.

SEE ALSO

`tcp`(4)

NAME

boottime, **time_second**, **time_uptime** — system time variables

SYNOPSIS

extern struct timeval boottime;
extern time_t time_uptime;
extern time_t time_second;

DESCRIPTION

The *time_second* variable is the system's "wall time" clock. It is set at boot by `inittodr`(9), and is updated by the `settimeofday`(2) system call and by periodic clock interrupts.

The *boottime* variable holds the system boot time. It is set from *time* at system boot, and is updated when the system time is adjusted with `settimeofday`(2).

The *time_uptime* variable is a monotonically increasing system clock. It is set from *time_second* at boot, and is updated by the periodic timer interrupt. (It is not updated by `settimeofday`(2).)

All of these variables contain times expressed in seconds and microseconds since midnight (0 hour), January 1, 1970.

Clock interrupts should be blocked when reading or writing *time_second* or *time_uptime*, because those variables are updated by **hardclock**(). *boottime* and *runtime* may be read and written without special precautions.

SEE ALSO

`settimeofday`(2), `hardclock`(9), `hz`(9), `inittodr`(9), `microtime`(9)

NAME

boottime, **time_second**, **time_uptime** — system time variables

SYNOPSIS

```
#include <sys/time.h>

extern struct timeval boottime;
extern time_t time_second;
extern time_t time_uptime;
```

DESCRIPTION

The *boottime* variable holds the system boot time.

The *time_second* variable is the system's "wall time" clock to the second.

The *time_uptime* variable is the number of seconds since boot.

The bintime(9), getbintime(9), microtime(9), getmicrotime(9), nanotime(9), and getnanotime(9) functions can be used to get the current time more accurately and in an atomic manner. Similarly, the binuptime(9), getbinuptime(9), microuptime(9), getmicrouptime(9), nanouptime(9), and getnanouptime(9) functions can be used to get the time elapse since boot more accurately and in an atomic manner. The *boottime* variable may be read and written without special precautions.

SEE ALSO

clock_settime(2), ntp_adjtime(2), settimeofday(2), bintime(9), binuptime(9), getbintime(9), getbinuptime(9), getmicrotime(9), getmicrouptime(9), getnanotime(9), getnanouptime(9), microtime(9), microuptime(9), nanotime(9), nanouptime(9)

Poul-Henning Kamp, "Timecounters: Efficient and precise timekeeping in SMP kernels", *Proceedings of EuroBSDCon 2002, Amsterdam.*

Marshall Kirk McKusick and George V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*, *Addison-Wesley*, 57-61,65-66, July 2004.

NAME

todr_attach, **todr_gettime**, **todr_settime**, **clock_ymdhms_to_secs**, **clock_secs_to_ymdhms** — time-of-day clock support

SYNOPSIS

```
#include <dev/clock_subr.h>

void
todr_attach(todr_chip_handle_t);

int
todr_gettime(todr_chip_handle_t, struct timeval *);

int
todr_settime(todr_chip_handle_t, struct timeval *);

void
clock_secs_to_ymdhms(int, struct clock_ymdhms *);

time_t
clock_ymdhms_to_secs(struct clock_ymdhms *);
```

DESCRIPTION

The **todr_***() functions provide an interface to read, set and control `time-of-day` devices. A driver for a `time-of-day` device registers its *todr_chip_handle_t* with machine-dependent code using the **todr_attach**() function. Alternatively, a machine-dependent front-end to a `time-of-day` device driver may obtain the *todr_chip_handle_t* directly.

The **todr_gettime**() retrieves the current data and time from the TODR device and returns it in the *struct timeval* storage provided by the caller. **todr_settime**() sets the date and time in the TODR device represented by *todr_chip_handle_t* according to the *struct timeval* argument.

The utilities **clock_secs_to_ymdhms**() and **clock_ymdhms_to_secs**() are provided to convert a time value in seconds to and from a structure representing the date and time as a ⟨year,month,day,weekday,hour,minute,seconds⟩ tuple. This structure is defined as follows:

```
struct clock_ymdhms {
        u_short dt_year;        /* Year */
        u_char dt_mon;          /* Month (1-12) */
        u_char dt_day;          /* Day (1-31) */
        u_char dt_wday;                 /* Day of week (0-6) */
        u_char dt_hour;                 /* Hour (0-23) */
        u_char dt_min;          /* Minute (0-59) */
        u_char dt_sec;          /* Second (0-59) */
};
```

Note: leap years are recognised by these conversion routines.

RETURN VALUES

The **todr_***() functions return 0 if the requested operation was successful; otherwise an error code from ⟨sys/errno.h⟩ shall be returned. However, behaviour is undefined if an invalid *todr_chip_handle_t* is passed to any of these functions.

The **clock_ymdhms_to_secs**() function returns −1 if the time in seconds would be less that zero or too large to fit in a *time_t*. The **clock_secs_to_ymdhms**() function never fails.

SEE ALSO

`intersil7170`(4), `mk48txx`(4), `inittodr`(9), `resettodr`(9), `time_second`(9)

NAME

ucom — interface for USB tty like devices

DESCRIPTION

The **ucom** driver is a (relatively) easy way to make a USB device look like a `tty`(4). It basically takes two bulk pipes, input and output, and makes a tty out of them. This is useful for a number of device types, e.g., serial ports (see `uftdi`(4)), modems (see `umodem`(4)), and devices that traditionally look like a tty (see `uvisor`(4)).

Communication between the real driver and the **ucom** driver is via the attachment arguments (when attached) and via the *ucom_methods* struct

ATTACHMENT

```
struct ucom_attach_args {
        int portno;
        int bulkin;
        int bulkout;
        u_int ibufsize;
        u_int ibufsizepad;
        u_int obufsize;
        u_int obufsizepad;
        usbd_device_handle device;
        usbd_interface_handle iface;
        struct ucom_methods *methods;
        void *arg;
};
```

`int portno`
: identifies the port if the devices should have more than one **ucom** attached. Use the value `UCOM_UNK_PORTNO` if there is only one port.

`int bulkin`
: the number of the bulk input pipe.

`int bulkout`
: the number of the bulk output pipe.

`u_int ibufsize`
: the size of the read requests on the bulk in pipe.

`u_int ibufsizepad`
: the size of the input buffer. This is usually the same as `ibufsize`.

`u_int obufsize`
: the size of the write requests on the bulk out pipe.

`u_int ibufsizepad`
: the size of the output buffer. This is usually the same as `obufsize`.

`usbd_device_handle device`
: a handle to the device.

usbd_interface_handle iface
: a handle to the interface that should be used.

struct ucom_methods *methods
: a pointer to the methods that the **ucom** driver should use for further communication with the driver.

void *arg
: the value that should be passed as first argument to each method.

METHODS

The `ucom_methods` struct contains a number of function pointers used by the **ucom** driver at various stages. If the device is not interested in being called at a particular point it should just use a `NULL` pointer and the **ucom** driver will use a sensible default.

```
struct ucom_methods {
        void (*ucom_get_status)(void *sc, int portno,
                                u_char *lsr, u_char *msr);
        void (*ucom_set)(void *sc, int portno, int reg, int onoff);
#define UCOM_SET_DTR 1
#define UCOM_SET_RTS 2
#define UCOM_SET_BREAK 3
        int (*ucom_param)(void *sc, int portno, struct termios *);
        int (*ucom_ioctl)(void *sc, int portno, u_long cmd,
                          void *data, int flag, struct lwp *l);
        int (*ucom_open)(void *sc, int portno);
        void (*ucom_close)(void *sc, int portno);
        void (*ucom_read)(void *sc, int portno, u_char **ptr,
                          uint32_t *count);
        void (*ucom_write)(void *sc, int portno, u_char *to,
                           u_char *from, uint32_t *count);
};
```

void (*ucom_get_status)(*void *sc, int portno, u_char *lsr, u_char *msr*)
: get the status of port *portno*. The status consists of the line status, *lsr*, and the modem status *msr*. The contents of these two bytes is exactly as for a 16550 UART.

void (*ucom_set)(*void *sc, int portno, int reg, int onoff*)
: Set (or unset) a particular feature of a port.

int (*ucom_param)(*void *sc, int portno, struct termios *t*)
: Set the speed, number of data bit, stop bits, and parity of a port according to the `termios`(4) struct.

int (*ucom_ioctl)(*void *sc, int portno, u_long cmd, void *data, int flag, struct lwp *l*)
: implements any non-standard `ioctl`(2) that a device needs.

int (*ucom_open)(*void *sc, int portno*)
: called just before the **ucom** driver opens the bulk pipes for the port.

void (*ucom_close)(*void *sc, int portno*)
: called just after the **ucom** driver closes the bulk pipes for the port.

void (*ucom_read)(*void *sc, int portno, u_char **ptr, uint32_t *count*)
: if the data delivered on the bulk pipe is not just the raw input characters this routine needs to adjust *ptr* and *count* so that they tell where to find the given number of raw characters.

void (*ucom_write)(*void *sc, int portno, u_char *dst, u_char *src, uint32_t *count*)
if the data written to the bulk pipe is not just the raw characters then this routine needs to copy *count* raw characters from *src* into the buffer at *dst* and do the appropriate padding. The *count* should be updated to the new size. The buffer at *src* is at most *ibufsize* bytes and the buffer at *dst* is *ibufsizepad* bytes.

Apart from these methods there is a function

void ucom_status_change(*struct ucom_softc **)

which should be called by the driver whenever it notices a status change.

SEE ALSO

tty(4), uftdi(4), umodem(4), usb(4), uvisor(4)

HISTORY

This **ucom** interface first appeared in NetBSD 1.5.

NAME

uiomove — move data described by a struct uio

SYNOPSIS

```
#include <sys/systm.h>

int
uiomove(void *buf, size_t n, struct uio *uio);
```

DESCRIPTION

The **uiomove** function copies up to *n* bytes between the kernel-space address pointed to by *buf* and the addresses described by *uio*, which may be in user-space or kernel-space.

The *uio* argument is a pointer to a *struct uio* as defined by ⟨sys/uio.h⟩:

```
struct uio {
        struct  iovec *uio_iov;/* pointer to array of iovecs */
        int     uio_iovcnt;    /* number of iovecs in array */
        off_t   uio_offset;    /* offset into file this uio corresponds to */
        size_t  uio_resid;     /* residual i/o count */
        enum    uio_rw uio_rw;
        struct  vmspace *uio_vmspace;
};
```

A *struct uio* typically describes data in motion. Several of the fields described below reflect that expectation.

uio_iov — Pointer to array of *struct iovecs*:

```
struct iovec {
        void   *iov_base;      /* Base address. */
        size_t  iov_len;       /* Length. */
};
```

uio_iovcnt — The number of iovecs in the array.

uio_offset — An offset into the corresponding object.

uio_resid — The amount of space described by the structure; notionally, the amount of data remaining to be transferred.

uio_rw — A flag indicating whether data should be read into the space (UIO_READ) or written from the space (UIO_WRITE).

uio_vmspace — A pointer to the address space which is being transferred to or from.

The value of *uio->uio_rw* controls whether **uiomove** copies data from *buf* to *uio* or vice versa.

The lesser of *n* or *uio->uio_resid* bytes are copied.

uiomove changes fields of the structure pointed to by *uio*, such that *uio->uio_resid* is decremented by the amount of data moved, *uio->uio_offset* is incremented by the same amount, and the array of iovecs is adjusted to point that much farther into the region described. This allows multiple calls to **uiomove** to easily be used to fill or drain the region of data.

RETURN VALUES

uiomove returns 0 on success or EFAULT if a bad address is encountered.

SEE ALSO

`copy`(9), `fetch`(9), `store`(9)

NAME

usbdi — USB device drivers interface

SYNOPSIS

```
#include <dev/usb/usb.h>
#include <dev/usb/usbdi.h>
```

DESCRIPTION

Device driver access to the USB bus centers around transfers. A transfer describes a communication with a USB device. A transfer is an abstract concept that can result in several physical packets being transferred to or from a device. A transfer is described by a *usbd_xfer_handle*. It is allocated by *usbd_alloc_xfer* and the data describing the transfer is filled by *usbd_setup_default_xfer* for control pipe transfers, by *usbd_setup_xfer* for bulk and interrupt transfers, and by *usbd_setup_isoc_xfer* for isochronous transfers.

describe *usbd_do_request*

describe pipes

describe usbd_status

Functions offered by usbdi

```
usbd_status usbd_open_pipe(usbd_interface_handle iface, uint8_t address,
        uint8_t flags,
        usbd_pipe_handle *pipe)

usbd_status usbd_close_pipe(usbd_pipe_handle pipe)

usbd_status usbd_transfer(usbd_xfer_handle req)

usbd_xfer_handle usbd_alloc_xfer(usbd_device_handle)

usbd_status usbd_free_xfer(usbd_xfer_handle xfer)

void usbd_setup_xfer(usbd_xfer_handle xfer, usbd_pipe_handle pipe,
        usbd_private_handle priv, void *buffer,
        uint32_t length, uint16_t flags, uint32_t timeout,
        usbd_callback)

void usbd_setup_default_xfer(usbd_xfer_handle xfer,
        usbd_device_handle dev,
        usbd_private_handle priv, uint32_t timeout,
        usb_device_request_t *req,  void *buffer,
        uint32_t length, uint16_t flags, usbd_callback)

void usbd_setup_isoc_xfer(usbd_xfer_handle xfer, usbd_pipe_handle pipe,
        usbd_private_handle priv, uint16_t *frlengths,
        uint32_t nframes, uint16_t flags, usbd_callback)

void   usbd_get_xfer_status(usbd_xfer_handle   xfer,   usbd_private_handle
       *priv,
        void **buffer, uint32_t *count, usbd_status *status)

usb_endpoint_descriptor_t
       *usbd_interface2endpoint_descriptor(usbd_interface_handle   iface,
       uint8_t address)
```

```
usbd_status usbd_abort_pipe(usbd_pipe_handle pipe)

usbd_status usbd_clear_endpoint_stall(usbd_pipe_handle pipe)

usbd_status usbd_clear_endpoint_stall_async(usbd_pipe_handle pipe)

usbd_status usbd_endpoint_count(usbd_interface_handle dev, uint8_t
        *count)

usbd_status usbd_interface_count(usbd_device_handle dev, uint8_t *count)

usbd_status usbd_interface2device_handle(usbd_interface_handle iface,
        usbd_device_handle *dev)

usbd_status usbd_device2interface_handle(usbd_device_handle dev, uint8_t
        ifaceno, usbd_interface_handle *iface)

usbd_device_handle usbd_pipe2device_handle(usbd_pipe_handle)

void *usbd_alloc_buffer(usbd_xfer_handle req, uint32_t size)

void usbd_free_buffer(usbd_xfer_handle req)

void *usbd_get_buffer(usbd_xfer_handle xfer)

usbd_status usbd_sync_transfer(usbd_xfer_handle req)

usbd_status usbd_open_pipe_intr(usbd_interface_handle iface, uint8_t
        address,
            uint8_t flags, usbd_pipe_handle *pipe,
            usbd_private_handle priv, void *buffer,
            uint32_t length, usbd_callback)

usbd_status usbd_do_request(usbd_device_handle pipe, usb_device_request_t
        *req, void *data)

usbd_status usbd_do_request_async(usbd_device_handle pipe,
        usb_device_request_t *req, void *data)

usbd_status usbd_do_request_flags(usbd_device_handle pipe,
        usb_device_request_t *req,
            void *data, uint16_t flags, int *)

usb_interface_descriptor_t
        *usbd_get_interface_descriptor(usbd_interface_handle iface)

usb_config_descriptor_t *usbd_get_config_descriptor(usbd_device_handle
        dev)

usb_device_descriptor_t *usbd_get_device_descriptor(usbd_device_handle
        dev)

usbd_status usbd_set_interface(usbd_interface_handle, int)

int usbd_get_no_alts(usb_config_descriptor_t *, int)

usbd_status usbd_get_interface(usbd_interface_handle iface, uint8_t
        *aiface)

void usbd_fill_deviceinfo (usbd_device_handle dev, struct usb_device_info
        *di)
```

```
int usbd_get_interface_altindex(usbd_interface_handle iface)

usb_interface_descriptor_t *usbd_find_idesc(usb_config_descriptor_t *cd,
       int iindex, int ano)

usb_endpoint_descriptor_t *usbd_find_edesc(usb_config_descriptor_t *cd,
       int ifaceidx, int altidx,
         int endptidx)

const char *usbd_errstr(usbd_status err)
```

Utilities from usbdi_util.h

Based on the routines in `usbdi.h` a number of utility functions have been defined that are accessible through `usbdi_util.h`

```
usbd_status usbd_get_desc(usbd_device_handle dev, int type,
                                    int index, int len, void *desc)

usbd_status usbd_get_config_desc(usbd_device_handle, int,
                                      usb_config_descriptor_t *)

usbd_status usbd_get_config_desc_full(usbd_device_handle, int,
                                      void *, int)

usbd_status usbd_get_device_desc(usbd_device_handle dev,
                                      usb_device_descriptor_t *d)

usbd_status usbd_set_address(usbd_device_handle dev, int addr)

usbd_status usbd_get_port_status(usbd_device_handle,
                                    int, usb_port_status_t *)

usbd_status usbd_set_hub_feature(usbd_device_handle dev, int)

usbd_status usbd_clear_hub_feature(usbd_device_handle, int)

usbd_status usbd_set_port_feature(usbd_device_handle dev, int, int)

usbd_status usbd_clear_port_feature(usbd_device_handle, int, int)

usbd_status usbd_get_device_status(usbd_device_handle,usb_status_t*)

usbd_status usbd_get_hub_status(usbd_device_handle dev,
                                      usb_hub_status_t *st)

usbd_status usbd_set_protocol(usbd_interface_handle dev, int report)

usbd_status usbd_get_report_descriptor
        (usbd_device_handle dev, int ifcno, int repid, int size, void *d)

struct usb_hid_descriptor *usbd_get_hid_descriptor
        (usbd_interface_handle ifc)

usbd_status usbd_set_report
        (usbd_interface_handle iface,int type,int id,void *data,int len)

usbd_status usbd_set_report_async
        (usbd_interface_handle iface,int type,int id,void *data,int len)

usbd_status usbd_get_report
        (usbd_interface_handle iface,int type,int id,void *data,int len)
```

`usbd_status usbd_set_idle`
(usbd_interface_handle iface, int duration, int id)

`usbd_status usbd_alloc_report_desc`
(usbd_interface_handle ifc, void **descp, int *sizep, int mem)

`usbd_status usbd_get_config`
(usbd_device_handle dev, uint8_t *conf)

`usbd_status usbd_get_string_desc`
(usbd_device_handle dev, int sindex, int langid, usb_string_descriptor_t *sdesc)

`void usbd_delay_ms(usbd_device_handle, u_int)`

`usbd_status usbd_set_config_no`
(usbd_device_handle dev, int no, int msg)

`usbd_status usbd_set_config_index`
(usbd_device_handle dev, int index, int msg)

`usbd_status usbd_bulk_transfer`
(usbd_xfer_handle xfer, usbd_pipe_handle pipe, uint16_t flags, uint32_t timeout, void *buf, uint32_t *size, char *lbl)

`void usb_detach_wait(device_ptr_t)`

`void usb_detach_wakeup(device_ptr_t)`

SEE ALSO

usb(4)

HISTORY

This **usbdi** interface first appeared in NetBSD 1.4. The interface is based on an early definition from the OpenUSBDI group within the USB organisation. Right after this definition the OpenUSBDI development got closed for open source developers, so this interface has not followed the further changes. The OpenUSBDI specification is now available again, but looks different.

BUGS

This man page is under development, so its biggest shortcoming is incompleteness.

NAME

userret — return path to user-mode execution

SYNOPSIS

```
#include <sys/lwp.h>
#include <sys/sched.h>

void
userret(struct lwp *l);
```

DESCRIPTION

The **userret**() function is executed after processing a trap (e.g., a system call or interrupt) before returning to user-mode execution. The implementation is machine dependent and is never invoked from machine-independent code. The function prototype for each architecture may be different to the prototype above, however the functionally provided by the **userret**() function on each architecture is essentially the same.

Specifically, the **userret**() function performs the following procedure:

- Detect a change in the signal disposition of the current process and invoke postsig(9) to post the signal to the process. This may occur when the outcome of the trap or syscall posted a signal to the process (e.g., invalid instruction trap).
- Check the *want_resched* flag to see if the scheduler requires the current process to be preempted by invoking preempt(9) (see cpu_need_resched(9)). This may occur if the clock interrupt causes the scheduler to determine that the current process has completed its time slice.
- Update the scheduler state.

SEE ALSO

cpu_need_resched(9), postsig(9), preempt(9), scheduler(9)

NAME

uvm — virtual memory system external interface

SYNOPSIS

```
#include <sys/param.h>
#include <uvm/uvm.h>
```

DESCRIPTION

The UVM virtual memory system manages access to the computer's memory resources. User processes and the kernel access these resources through UVM's external interface. UVM's external interface includes functions that:

- initialize UVM sub-systems
- manage virtual address spaces
- resolve page faults
- memory map files and devices
- perform uio-based I/O to virtual memory
- allocate and free kernel virtual memory
- allocate and free physical memory

In addition to exporting these services, UVM has two kernel-level processes: pagedaemon and swapper. The pagedaemon process sleeps until physical memory becomes scarce. When that happens, pagedaemon is awoken. It scans physical memory, paging out and freeing memory that has not been recently used. The swapper process swaps in runnable processes that are currently swapped out, if there is room.

There are also several miscellaneous functions.

INITIALIZATION

void
uvm_init(*void*);

void
uvm_init_limits(*struct lwp *l*);

void
uvm_setpagesize(*void*);

void
uvm_swap_init(*void*);

uvm_init() sets up the UVM system at system boot time, after the console has been setup. It initializes global state, the page, map, kernel virtual memory state, machine-dependent physical map, kernel memory allocator, pager and anonymous memory sub-systems, and then enables paging of kernel objects.

uvm_init_limits() initializes process limits for the named process. This is for use by the system startup for process zero, before any other processes are created.

uvm_setpagesize() initializes the uvmexp members pagesize (if not already done by machine-dependent code), pageshift and pagemask. It should be called by machine-dependent code early in the **pmap_init**() call (see pmap(9)).

uvm_swap_init() initializes the swap sub-system.

VIRTUAL ADDRESS SPACE MANAGEMENT

int
uvm_map(*struct vm_map *map*, *vaddr_t *startp*, *vsize_t size*, *struct uvm_object *uobj*, *voff_t uoffset*, *vsize_t align*, *uvm_flag_t flags*);

void
uvm_unmap(*struct vm_map *map*, *vaddr_t start*, *vaddr_t end*);

int
uvm_map_pageable(*struct vm_map *map*, *vaddr_t start*, *vaddr_t end*, *bool new_pageable*, *int lockflags*);

bool
uvm_map_checkprot(*struct vm_map *map*, *vaddr_t start*, *vaddr_t end*, *vm_prot_t protection*);

int
uvm_map_protect(*struct vm_map *map*, *vaddr_t start*, *vaddr_t end*, *vm_prot_t new_prot*, *bool set_max*);

int
uvm_deallocate(*struct vm_map *map*, *vaddr_t start*, *vsize_t size*);

*struct vmspace **
uvmspace_alloc(*vaddr_t min*, *vaddr_t max*, *int pageable*);

void
uvmspace_exec(*struct lwp *l*, *vaddr_t start*, *vaddr_t end*);

*struct vmspace **
uvmspace_fork(*struct vmspace *vm*);

void
uvmspace_free(*struct vmspace *vm1*);

void
uvmspace_share(*struct proc *p1*, *struct proc *p2*);

void
uvmspace_unshare(*struct lwp *l*);

bool
uvm_uarea_alloc(*vaddr_t *uaddrp*);

void
uvm_uarea_free(*vaddr_t uaddr*);

uvm_map() establishes a valid mapping in map *map*, which must be unlocked. The new mapping has size *size*, which must be a multiple of PAGE_SIZE. The *uobj* and *uoffset* arguments can have four meanings. When *uobj* is NULL and *uoffset* is UVM_UNKNOWN_OFFSET, **uvm_map**() does not use the machine-dependent PMAP_PREFER function. If *uoffset* is any other value, it is used as the hint to PMAP_PREFER. When *uobj* is not NULL and *uoffset* is UVM_UNKNOWN_OFFSET, **uvm_map**() finds the offset based upon the virtual address, passed as *startp*. If *uoffset* is any other value, we are doing a normal mapping at this offset. The start address of the map will be returned in *startp*.

align specifies alignment of mapping unless UVM_FLAG_FIXED is specified in *flags*. *align* must be a power of 2.

flags passed to **uvm_map**() are typically created using the **UVM_MAPFLAG**(*vm_prot_t prot*, *vm_prot_t maxprot*, *vm_inherit_t inh*, *int advice*, *int flags*) macro, which uses the following values. The *prot* and *maxprot* can take are:

```
#define UVM_PROT_MASK   0x07    /* protection mask */
#define UVM_PROT_NONE   0x00    /* protection none */
#define UVM_PROT_ALL    0x07    /* everything */
```

```
#define UVM_PROT_READ   0x01    /* read */
#define UVM_PROT_WRITE  0x02    /* write */
#define UVM_PROT_EXEC   0x04    /* exec */
#define UVM_PROT_R      0x01    /* read */
#define UVM_PROT_W      0x02    /* write */
#define UVM_PROT_RW     0x03    /* read-write */
#define UVM_PROT_X      0x04    /* exec */
#define UVM_PROT_RX     0x05    /* read-exec */
#define UVM_PROT_WX     0x06    /* write-exec */
#define UVM_PROT_RWX    0x07    /* read-write-exec */
```

The values that *inh* can take are:

```
#define UVM_INH_MASK    0x30    /* inherit mask */
#define UVM_INH_SHARE   0x00    /* "share" */
#define UVM_INH_COPY    0x10    /* "copy" */
#define UVM_INH_NONE    0x20    /* "none" */
#define UVM_INH_DONATE  0x30    /* "donate" << not used */
```

The values that *advice* can take are:

```
#define UVM_ADV_NORMAL     0x0  /* 'normal' */
#define UVM_ADV_RANDOM     0x1  /* 'random' */
#define UVM_ADV_SEQUENTIAL 0x2  /* 'sequential' */
#define UVM_ADV_MASK       0x7  /* mask */
```

The values that *flags* can take are:

```
#define UVM_FLAG_FIXED   0x010000 /* find space */
#define UVM_FLAG_OVERLAY 0x020000 /* establish overlay */
#define UVM_FLAG_NOMERGE 0x040000 /* don't merge map entries */
#define UVM_FLAG_COPYONW 0x080000 /* set copy_on_write flag */
#define UVM_FLAG_AMAPPAD 0x100000 /* for bss: pad amap to reduce malloc() */
#define UVM_FLAG_TRYLOCK 0x200000 /* fail if we can not lock map */
```

The UVM_MAPFLAG macro arguments can be combined with an or operator. There are several special purpose macros for checking protection combinations, e.g., the UVM_PROT_WX macro. There are also some additional macros to extract bits from the flags. The UVM_PROTECTION, UVM_INHERIT, UVM_MAXPROTECTION and UVM_ADVICE macros return the protection, inheritance, maximum protection and advice, respectively. **uvm_map**() returns a standard UVM return value.

uvm_unmap() removes a valid mapping, from *start* to *end*, in map *map*, which must be unlocked.

uvm_map_pageable() changes the pageability of the pages in the range from *start* to *end* in map *map* to *new_pageable*. **uvm_map_pageable**() returns a standard UVM return value.

uvm_map_checkprot() checks the protection of the range from *start* to *end* in map *map* against *protection*. This returns either true or false.

uvm_map_protect() changes the protection *start* to *end* in map *map* to *new_prot*, also setting the maximum protection to the region to *new_prot* if *set_max* is non-zero. This function returns a standard UVM return value.

uvm_deallocate() deallocates kernel memory in map *map* from address *start* to *start + size*.

uvmspace_alloc() allocates and returns a new address space, with ranges from *min* to *max*, setting the pageability of the address space to *pageable*.

uvmspace_exec() either reuses the address space of lwp *l* if there are no other references to it, or creates a new one with **uvmspace_alloc**(). The range of valid addresses in the address space is reset to *start* through *end*.

uvmspace_fork() creates and returns a new address space based upon the *vm1* address space, typically used when allocating an address space for a child process.

uvmspace_free() lowers the reference count on the address space *vm*, freeing the data structures if there are no other references.

uvmspace_share() causes process p2 to share the address space of *p1*.

uvmspace_unshare() ensures that lwp *l* has its own, unshared address space, by creating a new one if necessary by calling **uvmspace_fork**().

uvm_uarea_alloc() allocates virtual space for a u-area (i.e., a kernel stack) and stores its virtual address in ∗*uaddrp*. The return value is true if the u-area is already backed by wired physical memory, otherwise false.

uvm_uarea_free() frees a u-area allocated with **uvm_uarea_alloc**(), freeing both the virtual space and any physical pages which may have been allocated to back that virtual space later.

PAGE FAULT HANDLING

int
uvm_fault(*struct vm_map ∗orig_map*, *vaddr_t vaddr*, *vm_prot_t access_type*);

uvm_fault() is the main entry point for faults. It takes *orig_map* as the map the fault originated in, a *vaddr* offset into the map the fault occurred, and *access_type* describing the type of access requested. **uvm_fault**() returns a standard UVM return value.

MEMORY MAPPING FILES AND DEVICES

void
uvm_vnp_setsize(*struct vnode ∗vp*, *voff_t newsize*);

void ∗
ubc_alloc(*struct uvm_object ∗uobj*, *voff_t offset*, *vsize_t ∗lenp*, *int advice*, *int flags*);

void
ubc_release(*void ∗va*, *int flags*);

int
ubc_uiomove(*struct uvm_object ∗uobj*, *struct uio ∗uio*, *vsize_t todo*, *int advice*, *int flags*);

uvm_vnp_setsize() sets the size of vnode *vp* to *newsize*. Caller must hold a reference to the vnode. If the vnode shrinks, pages no longer used are discarded.

ubc_alloc() creates a kernel mapping of *uobj* starting at offset *offset*. The desired length of the mapping is pointed to by *lenp*, but the actual mapping may be smaller than this. *lenp* is updated to contain the actual length mapped. *advice* is the access pattern hint, which must be one of

UVM_ADV_NORMAL	No hint
UVM_ADV_RANDOM	Random access hint
UVM_ADV_SEQUENTIAL	Sequential access hint (from lower offset to higher offset)

The possible *flags* are

UBC_READ	Mapping will be accessed for read.
UBC_WRITE	Mapping will be accessed for write.
UBC_FAULTBUSY	Fault in window's pages already during mapping operation. Makes sense only for write.

Currently, *uobj* must actually be a vnode object. Once the mapping is created, it must be accessed only by methods that can handle faults, such as **uiomove**() or **kcopy**(). Page faults on the mapping will result in the vnode's **VOP_GETPAGES**() method being called to resolve the fault.

ubc_release() frees the mapping at *va* for reuse. The mapping may be cached to speed future accesses to the same region of the object. The flags can be any of

UBC_UNMAP	Do not cache mapping.

ubc_uiomove() allocates an UBC memory window, performs I/O on it and unmaps the window. The *advice* parameter takes the same values as the respective parameter in **ubc_alloc**() and the *flags* parameter takes the same arguments as **ubc_alloc**() and **ubc_unmap**(). Additionally, the flag UBC_PARTIALOK can be provided to indicate that it is acceptable to return if an error occurs mid-transfer.

VIRTUAL MEMORY I/O

int
uvm_io(*struct vm_map *map*, *struct uio *uio*);

uvm_io() performs the I/O described in *uio* on the memory described in *map*.

ALLOCATION OF KERNEL MEMORY

vaddr_t
uvm_km_alloc(*struct vm_map *map*, *vsize_t size*, *vsize_t align*, *uvm_flag_t flags*);

void
uvm_km_free(*struct vm_map *map*, *vaddr_t addr*, *vsize_t size*, *uvm_flag_t flags*);

*struct vm_map **
uvm_km_suballoc(*struct vm_map *map*, *vaddr_t *min*, *vaddr_t *max*, *vsize_t size*, *bool pageable*, *bool fixed*, *struct vm_map *submap*);

uvm_km_alloc() allocates *size* bytes of kernel memory in map *map*. The first address of the allocated memory range will be aligned according to the *align* argument (specify 0 if no alignment is necessary). The alignment must be a multiple of page size. The *flags* is a bitwise inclusive OR of the allocation type and operation flags.

The allocation type should be one of:

UVM_KMF_WIRED Wired memory.

UVM_KMF_PAGEABLE
Demand-paged zero-filled memory.

UVM_KMF_VAONLY Virtual address only. No physical pages are mapped in the allocated region. If necessary, it's the caller's responsibility to enter page mappings. It's also the caller's responsibility to clean up the mappings before freeing the address range.

The following operation flags are available:

UVM_KMF_CANFAIL
: Can fail even if `UVM_KMF_NOWAIT` is not specified and `UVM_KMF_WAITVA` is specified.

UVM_KMF_ZERO
: Request zero-filled memory. Only supported for `UVM_KMF_WIRED`. Shouldn't be used with other types.

UVM_KMF_TRYLOCK
: Fail if we can't lock the map.

UVM_KMF_NOWAIT
: Fail immediately if no memory is available.

UVM_KMF_WAITVA
: Sleep to wait for the virtual address resources if needed.

(If neither `UVM_KMF_NOWAIT` nor `UVM_KMF_CANFAIL` are specified and `UVM_KMF_WAITVA` is specified, **uvm_km_alloc**() will never fail, but rather sleep indefinitely until the allocation succeeds.)

Pageability of the pages allocated with `UVM_KMF_PAGEABLE` can be changed by **uvm_map_pageable**(). In that case, the entire range must be changed atomically. Changing a part of the range is not supported.

uvm_km_free() frees the memory range allocated by **uvm_km_alloc**(). *addr* must be an address returned by **uvm_km_alloc**(). *map* and *size* must be the same as the ones used for the corresponding **uvm_km_alloc**(). *flags* must be the allocation type used for the corresponding **uvm_km_alloc**().

uvm_km_free() is the only way to free memory ranges allocated by **uvm_km_alloc**(). **uvm_unmap**() must not be used.

uvm_km_suballoc() allocates submap from *map*, creating a new map if *submap* is `NULL`. The addresses of the submap can be specified exactly by setting the *fixed* argument to non-zero, which causes the *min* argument to specify the beginning of the address in the submap. If *fixed* is zero, any address of size *size* will be allocated from *map* and the start and end addresses returned in *min* and *max*. If *pageable* is non-zero, entries in the map may be paged out.

ALLOCATION OF PHYSICAL MEMORY

*struct vm_page **
uvm_pagealloc(*struct uvm_object *uobj*, *voff_t off*, *struct vm_anon *anon*, *int flags*);

void
uvm_pagerealloc(*struct vm_page *pg*, *struct uvm_object *newobj*, *voff_t newoff*);

void
uvm_pagefree(*struct vm_page *pg*);

int
uvm_pglistalloc(*psize_t size*, *paddr_t low*, *paddr_t high*, *paddr_t alignment*, *paddr_t boundary*, *struct pglist *rlist*, *int nsegs*, *int waitok*);

void
uvm_pglistfree(*struct pglist *list*);

void
uvm_page_physload(*vaddr_t start*, *vaddr_t end*, *vaddr_t avail_start*, *vaddr_t avail_end*, *int free_list*);

uvm_pagealloc() allocates a page of memory at virtual address *off* in either the object *uobj* or the anonymous memory *anon*, which must be locked by the caller. Only one of *uobj* and *anon* can be non `NULL`. Returns `NULL` when no page can be found. The flags can be any of

```
#define UVM_PGA_USERESERVE      0x0001  /* ok to use reserve pages */
#define UVM_PGA_ZERO            0x0002  /* returned page must be zero'd */
```

UVM_PGA_USERESERVE means to allocate a page even if that will result in the number of free pages being lower than uvmexp.reserve_pagedaemon (if the current thread is the pagedaemon) or uvmexp.reserve_kernel (if the current thread is not the pagedaemon). UVM_PGA_ZERO causes the returned page to be filled with zeroes, either by allocating it from a pool of pre-zeroed pages or by zeroing it in-line as necessary.

uvm_pagerealloc() reallocates page *pg* to a new object *newobj*, at a new offset *newoff*.

uvm_pagefree() frees the physical page *pg*. If the content of the page is known to be zero-filled, caller should set PG_ZERO in pg->flags so that the page allocator will use the page to serve future UVM_PGA_ZERO requests efficiently.

uvm_pglistalloc() allocates a list of pages for size *size* byte under various constraints. *low* and *high* describe the lowest and highest addresses acceptable for the list. If *alignment* is non-zero, it describes the required alignment of the list, in power-of-two notation. If *boundary* is non-zero, no segment of the list may cross this power-of-two boundary, relative to zero. *nsegs* is the maximum number of physically contiguous segments. If *waitok* is non-zero, the function may sleep until enough memory is available. (It also may give up in some situations, so a non-zero *waitok* does not imply that **uvm_pglistalloc**() cannot return an error.) The allocated memory is returned in the *rlist* list; the caller has to provide storage only, the list is initialized by **uvm_pglistalloc**().

uvm_pglistfree() frees the list of pages pointed to by *list*. If the content of the page is known to be zero-filled, caller should set PG_ZERO in pg->flags so that the page allocator will use the page to serve future UVM_PGA_ZERO requests efficiently.

uvm_page_physload() loads physical memory segments into VM space on the specified *free_list*. It must be called at system boot time to set up physical memory management pages. The arguments describe the *start* and *end* of the physical addresses of the segment, and the available start and end addresses of pages not already in use.

PROCESSES

void
uvm_pageout(*void*);

void
uvm_scheduler(*void*);

void
uvm_swapin(*struct lwp *l*);

uvm_pageout() is the main loop for the page daemon.

uvm_scheduler() is the process zero main loop, which is to be called after the system has finished starting other processes. It handles the swapping in of runnable, swapped out processes in priority order.

uvm_swapin() swaps in the named lwp.

PAGE LOAN

int
uvm_loan(*struct vm_map *map*, *vaddr_t start*, *vsize_t len*, *void *v*, *int flags*);

void
uvm_unloan(*void *v*, *int npages*, *int flags*);

uvm_loan() loans pages in a map out to anons or to the kernel. *map* should be unlocked, *start* and *len* should be multiples of PAGE_SIZE. Argument *flags* should be one of

```
#define UVM_LOAN_TOANON       0x01    /* loan to anons */
#define UVM_LOAN_TOPAGE       0x02    /* loan to kernel */
```

v should be pointer to array of pointers to `struct anon` or `struct vm_page`, as appropriate. The caller has to allocate memory for the array and ensure it's big enough to hold *len / PAGE_SIZE* pointers. Returns 0 for success, or appropriate error number otherwise. Note that wired pages can't be loaned out and **uvm_loan**() will fail in that case.

uvm_unloan() kills loans on pages or anons. The *v* must point to the array of pointers initialized by previous call to **uvm_loan**(). *npages* should match number of pages allocated for loan, this also matches number of items in the array. Argument *flags* should be one of

```
#define UVM_LOAN_TOANON       0x01    /* loan to anons */
#define UVM_LOAN_TOPAGE       0x02    /* loan to kernel */
```

and should match what was used for previous call to **uvm_loan**().

MISCELLANEOUS FUNCTIONS

*struct uvm_object **
uao_create(*vsize_t size*, *int flags*);

void
uao_detach(*struct uvm_object *uobj*);

void
uao_reference(*struct uvm_object *uobj*);

bool
uvm_chgkprot(*void *addr*, *size_t len*, *int rw*);

void
uvm_kernacc(*void *addr*, *size_t len*, *int rw*);

int
uvm_vslock(*struct vmspace *vs*, *void *addr*, *size_t len*, *vm_prot_t prot*);

void
uvm_vsunlock(*struct vmspace *vs*, *void *addr*, *size_t len*);

void
uvm_meter(*void*);

void
uvm_fork(*struct lwp *l1*, *struct lwp *l2*, *bool shared*);

int
uvm_grow(*struct proc *p*, *vaddr_t sp*);

void
uvn_findpages(*struct uvm_object *uobj*, *voff_t offset*, *int *npagesp*, *struct vm_page **pps*, *int flags*);

void
uvm_swap_stats(*int cmd*, *struct swapent *sep*, *int sec*, *register_t *retval*);

The **uao_create**(), **uao_detach**(), and **uao_reference**() functions operate on anonymous memory objects, such as those used to support System V shared memory. **uao_create**() returns an object of size

size with flags:

```
#define UAO_FLAG_KERNOBJ        0x1     /* create kernel object */
#define UAO_FLAG_KERNSWAP       0x2     /* enable kernel swap */
```

which can only be used once each at system boot time. **uao_reference**() creates an additional reference to the named anonymous memory object. **uao_detach**() removes a reference from the named anonymous memory object, destroying it if removing the last reference.

uvm_chgkprot() changes the protection of kernel memory from *addr* to *addr + len* to the value of *rw*. This is primarily useful for debuggers, for setting breakpoints. This function is only available with options KGDB.

uvm_kernacc() checks the access at address *addr* to *addr + len* for *rw* access in the kernel address space.

uvm_vslock() and **uvm_vsunlock**() control the wiring and unwiring of pages for process *p* from *addr* to *addr + len*. These functions are normally used to wire memory for I/O.

uvm_meter() calculates the load average and wakes up the swapper if necessary.

uvm_fork() forks a virtual address space for process' (old) *p1* and (new) *p2*. If the *shared* argument is non zero, p1 shares its address space with p2, otherwise a new address space is created. This function currently has no return value, and thus cannot fail. In the future, this function will be changed to allow it to fail in low memory conditions.

uvm_grow() increases the stack segment of process *p* to include *sp*.

uvn_findpages() looks up or creates pages in *uobj* at offset *offset*, marks them busy and returns them in the *pps* array. Currently *uobj* must be a vnode object. The number of pages requested is pointed to by *npagesp*, and this value is updated with the actual number of pages returned. The flags can be

```
#define UFP_ALL         0x00    /* return all pages requested */
#define UFP_NOWAIT      0x01    /* don't sleep */
#define UFP_NOALLOC     0x02    /* don't allocate new pages */
#define UFP_NOCACHE     0x04    /* don't return pages which already exist */
#define UFP_NORDONLY    0x08    /* don't return PG_READONLY pages */
```

UFP_ALL is a pseudo-flag meaning all requested pages should be returned. UFP_NOWAIT means that we must not sleep. UFP_NOALLOC causes any pages which do not already exist to be skipped. UFP_NOCACHE causes any pages which do already exist to be skipped. UFP_NORDONLY causes any pages which are marked PG_READONLY to be skipped.

uvm_swap_stats() implements the SWAP_STATS and SWAP_OSTATS operation of the swapctl(2) system call. *cmd* is the requested command, SWAP_STATS or SWAP_OSTATS. The function will copy no more than *sec* entries in the array pointed by *sep*. On return, *retval* holds the actual number of entries copied in the array.

SYSCTL

UVM provides support for the CTL_VM domain of the sysctl(3) hierarchy. It handles the VM_LOADAVG, VM_METER, VM_UVMEXP, and VM_UVMEXP2 nodes, which return the current load averages, calculates current VM totals, returns the uvmexp structure, and a kernel version independent view of the uvmexp structure, respectively. It also exports a number of tunables that control how much VM space is allowed to be consumed by various tasks. The load averages are typically accessed from userland using the getloadavg(3) function. The uvmexp structure has all global state of the UVM system, and has the following members:

```
/* vm_page constants */
int pagesize;   /* size of a page (PAGE_SIZE): must be power of 2 */
```

```
int pagemask;   /* page mask */
int pageshift;  /* page shift */

/* vm_page counters */
int npages;     /* number of pages we manage */
int free;       /* number of free pages */
int active;     /* number of active pages */
int inactive;   /* number of pages that we free'd but may want back */
int paging;     /* number of pages in the process of being paged out */
int wired;      /* number of wired pages */
int reserve_pagedaemon; /* number of pages reserved for pagedaemon */
int reserve_kernel; /* number of pages reserved for kernel */

/* pageout params */
int freemin;    /* min number of free pages */
int freetarg;   /* target number of free pages */
int inactarg;   /* target number of inactive pages */
int wiredmax;   /* max number of wired pages */

/* swap */
int nswapdev;   /* number of configured swap devices in system */
int swpages;    /* number of PAGE_SIZE'ed swap pages */
int swpginuse;  /* number of swap pages in use */
int nswget;     /* number of times fault calls uvm_swap_get() */
int nanon;      /* number total of anon's in system */
int nfreeanon;  /* number of free anon's */

/* stat counters */
int faults;             /* page fault count */
int traps;              /* trap count */
int intrs;              /* interrupt count */
int swtch;              /* context switch count */
int softs;              /* software interrupt count */
int syscalls;           /* system calls */
int pageins;            /* pagein operation count */
                        /* pageouts are in pdpageouts below */
int swapins;            /* swapins */
int swapouts;           /* swapouts */
int pgswapin;           /* pages swapped in */
int pgswapout;          /* pages swapped out */
int forks;              /* forks */
int forks_ppwait;       /* forks where parent waits */
int forks_sharevm;      /* forks where vmspace is shared */

/* fault subcounters */
int fltnoram;   /* number of times fault was out of ram */
int fltnoanon;  /* number of times fault was out of anons */
int fltpgwait;  /* number of times fault had to wait on a page */
int fltpgrele;  /* number of times fault found a released page */
int fltrelck;   /* number of times fault relock called */
int fltrelckok; /* number of times fault relock is a success */
int fltanget;   /* number of times fault gets anon page */
```

```
int fltanretry; /* number of times fault retrys an anon get */
int fltamcopy;  /* number of times fault clears "needs copy" */
int fltnamap;   /* number of times fault maps a neighbor anon page */
int fltnomap;   /* number of times fault maps a neighbor obj page */
int fltlget;    /* number of times fault does a locked pgo_get */
int fltget;     /* number of times fault does an unlocked get */
int flt_anon;   /* number of times fault anon (case 1a) */
int flt_acow;   /* number of times fault anon cow (case 1b) */
int flt_obj;    /* number of times fault is on object page (2a) */
int flt_prcopy; /* number of times fault promotes with copy (2b) */
int flt_przero; /* number of times fault promotes with zerofill (2b) */

/* daemon counters */
int pdwoke;     /* number of times daemon woke up */
int pdrevs;     /* number of times daemon rev'd clock hand */
int pdswout;    /* number of times daemon called for swapout */
int pdfreed;    /* number of pages daemon freed since boot */
int pdscans;    /* number of pages daemon scanned since boot */
int pdanscan;   /* number of anonymous pages scanned by daemon */
int pdobscan;   /* number of object pages scanned by daemon */
int pdreact;    /* number of pages daemon reactivated since boot */
int pdbusy;     /* number of times daemon found a busy page */
int pdpageouts; /* number of times daemon started a pageout */
int pdpending;  /* number of times daemon got a pending pageout */
int pddeact;    /* number of pages daemon deactivates */
```

NOTES

uvm_chgkprot() is only available if the kernel has been compiled with options KGDB.

All structure and types whose names begin with "vm_" will be renamed to "uvm_".

SEE ALSO

swapctl(2), getloadavg(3), kvm(3), sysctl(3), ddb(4), options(4), memoryallocators(9), pmap(9)

HISTORY

UVM is a new VM system developed at Washington University in St. Louis (Missouri). UVM's roots lie partly in the Mach-based 4.4BSD VM system, the FreeBSD VM system, and the SunOS 4 VM system. UVM's basic structure is based on the 4.4BSD VM system. UVM's new anonymous memory system is based on the anonymous memory system found in the SunOS 4 VM (as described in papers published by Sun Microsystems, Inc.). UVM also includes a number of features new to BSD including page loanout, map entry passing, simplified copy-on-write, and clustered anonymous memory pageout. UVM is also further documented in an August 1998 dissertation by Charles D. Cranor.

UVM appeared in NetBSD 1.4.

AUTHORS

Charles D. Cranor ⟨chuck@ccrc.wustl.edu⟩ designed and implemented UVM.

Matthew Green ⟨mrg@eterna.com.au⟩ wrote the swap-space management code and handled the logistical issues involved with merging UVM into the NetBSD source tree.

Chuck Silvers ⟨chuq@chuq.com⟩ implemented the aobj pager, thus allowing UVM to support System V shared memory and process swapping. He also designed and implemented the UBC part of UVM, which uses UVM pages to cache vnode data rather than the traditional buffer cache buffers.

NAME

vattr, **vattr_null**, **VATTR_NULL** — vnode attributes

SYNOPSIS

```
#include <sys/param.h>
#include <sys/vnode.h>

void
vattr_null(struct vattr *vap);

void
VATTR_NULL(struct vattr *vap);
```

DESCRIPTION

Vnode attributes describe attributes of a file or directory including file permissions, owner, group, size, access time and modification time.

A vnode attribute has the following structure:

```
struct vattr {
        enum vtype      va_type;        /* vnode type (for create) */
        mode_t          va_mode;        /* files access mode and type */
        nlink_t         va_nlink;       /* number of references to file */
        uid_t           va_uid;         /* owner user id */
        gid_t           va_gid;         /* owner group id */
        long            va_fsid;        /* file system id (dev for now) */
        long            va_fileid;      /* file id */
        u_quad_t        va_size;        /* file size in bytes */
        long            va_blocksize;   /* blocksize preferred for i/o */
        struct timespec va_atime;       /* time of last access */
        struct timespec va_mtime;       /* time of last modification */
        struct timespec va_ctime;       /* time file changed */
        struct timespec va_birthtime;   /* time file created */
        u_long          va_gen;         /* generation number of file */
        u_long          va_flags;       /* flags defined for file */
        dev_t           va_rdev;        /* device the special file represents */
        u_quad_t        va_bytes;       /* bytes of disk space held by file */
        u_quad_t        va_filerev;     /* file modification number */
        u_int           va_vaflags;     /* operations flags, see below */
        long            va_spare;       /* remain quad aligned */
};
```

A field value of VNOVAL represents a field whose value is unavailable or which is not to be changed. Valid flag values for *va_flags* are:

VA_UTIMES_NULL
: utimes argument was NULL

VA_EXCLUSIVE
: exclusive create request

Vnode attributes for a file are set by the vnode operation VOP_SETATTR(9). Vnode attributes for a file are retrieved by the vnode operation VOP_GETATTR(9). For more information on vnode operations see vnodeops(9).

FUNCTIONS

vattr_null(*vap*)
: Set vnode attributes in *vap* to VNOVAL.

VATTR_NULL(*vap*)
: This function is an alias for **vattr_null**().

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the vnode attributes can be found. All pathnames are relative to `/usr/src`.

The vnode attributes ares implemented within the file `sys/kern/vfs_subr2.c`.

SEE ALSO

`intro`(9), `vfs`(9), `vnode`(9), `vnodeops`(9)

NAME

vcons — generic virtual console framework

SYNOPSIS

```
#include <wscons/wsdisplay_vconsvar.h>

int
vcons_init(struct vcons_data *vd, void *cookie,
    struct wsscreen_descr *desc, struct wsdisplay_accessops *accops);

int
vcons_init_screen(struct vcons_data *vd, struct vcons_screen *scr,
    int exists, long *defattr);

void
vcons_redraw_screen(struct vcons_screen *scr);
```

DESCRIPTION

These functions are used to setup and control the generic virtual console framework.

The **vcons_init**() function initializes the framework, it needs to be called for each driver that's going to use **vcons**.

vcons_init_screen() adds a virtual screen to a display.

vcons_redraw_screen() redraws a screen. A driver should call it when returning to terminal emulation mode, for instance when X exits.

struct vcons_data contains all information needed to manage virtual consoles on a display, usually it will be a member of the driver's softc.

struct vcons_screen describes a virtual screen.

USAGE

To use vcons with a driver it needs to be initialized by calling **vcons_init**(), usually in the driver's attach function.

vd should be a pointer to the driver's *struct vcons_data*.

cookie should be a pointer to the driver's softc.

desc should point to a *struct wsscreen_descr* describing the default screen type for this display.

accops points to the driver's *struct wsdisplay_accessops* so **vcons_init**() can fill it in with its own implementations of **alloc_screen**(), **free_screen**(), and **show_screen**().

A driver should however provide its own **ioctl**() and **mmap**() implementations. Both will receive a pointer to the driver's *struct vcons_data* as first parameter.

After initialization the driver needs to provide a callback function that will be called whenever a screen is added. Its purpose is to set up the *struct rasops_info* describing the screen. After that the drawing methods in *struct rasops_info* will be replaced with wrappers which call the original drawing functions (which may or may not be provided by the driver) only when the respective screen is visible. To add a virtual screen the driver one should call **vcons_init_screen**() which will call the callback function described above, allocate storage for characters and attributes based on whatever the callback set up in *struct rasops_info*, and add the screen to a list kept in *struct vcons_data*.

The callback needs to have this form:

void **init_screen**(*void *cookie*, *struct vcons_screen *scr*, *int existing*, *long *defattr*)

and should be stored in the *init_screen* member found in *struct vcons_data*. The arguments are:

cookie is the cookie passed to **vcons_init**()

scr points to the *struct vcons_screen* being added, its *scr_ri* member, a *struct rasops_info*, needs to be filled in.

existing is non-zero if the screen already exists and is only added to the list.

defattr points to the screen's default text attribute. It's filled in by **vcons_init_screen**() by calling the **alloc_attr**() method found in *struct rasops_info*.

When attaching a wsdisplay(9) the *accesscookie* member of the *struct wsemuldisplaydev_attach_args* passed to **config_found**() needs to be a pointer to the driver's *struct vcons_data*.

The following members of *struct vcons_screen* may be of interest to drivers:

scr_ri contains the *struct rasops_info* describing the screen's geometry, access methods and so on.

scr_cookie the value passed as cookie to **vcons_init**(). Usually the driver's softc.

scr_vd the driver's *struct vcons_data*.

scr_flags can be zero or any combination of:

- VCONS_NO_REDRAW — don't call **vcons_redraw_screen**() when this screen becomes visible.
- VCONS_SCREEN_IS_STATIC — don't free(9) this screen's *struct vcons_screen* in **free_screen**() - useful if the screen has been statically allocated.

scr_status currently contains only one flag, VCONS_IS_VISIBLE, which is set when the screen is visible.

SEE ALSO

wscons(4), wsdisplay(4)

NAME

veriexec — in-kernel file integrity subsystem KPI

SYNOPSIS

```
#include <sys/verified_exec.h>
```

DESCRIPTION

veriexec is the KPI for *Veriexec*, the NetBSD in-kernel file integrity subsystem. It is responsible for managing the supported hashing algorithms, fingerprint calculation and comparison, file monitoring tables, and relevant hooks to enforce the *Veriexec* policy.

Core Routines

void **veriexec_init**(*void*)

Initialize the *Veriexec* subsystem. Called only once during system startup.

bool **veriexec_lookup**(*struct vnode *vp*)

Check if *vp* is monitored by *Veriexec* or not. Returns `true` if it is, or `false` otherwise.

int **veriexec_verify**(*struct lwp *l*, *struct vnode *vp*, *const u_char *name*, *int flag*, *bool *found*)

Verifies the digital fingerprint of *vp*. *name* is the filename, and *flag* is the access flag. The access flag can be one of:

`VERIEXEC_DIRECT`

The file was executed directly via `execve`(2).

`VERIEXEC_INDIRECT`

The file was executed indirectly, either as an interpreter for a script or mapped to an executable memory region.

`VERIEXEC_FILE`

The file was opened for reading/writing.

l is the LWP for the request context.

An optional argument, *found*, is a pointer to a boolean indicating whether an entry for the file was found in the *Veriexec* tables.

void **veriexec_purge**(*struct vnode *vp*)

Purge the file entry for *vp*. This invalidates the fingerprint so it will be evaluated next time the file is accessed.

Fingerprint Related Routines

veriexec_fpops_add(*const char *fp_type*, *size_t hash_len*, *size_t ctx_size*, *veriexec_fpop_init_t init*, *veriexec_fpop_update_t update*, *veriexec_fpop_final_t final*)

Add support for fingerprinting algorithm *fp_type* with binary hash length *hash_len* and calculation context size *ctx_size* to *Veriexec*. *init*, *update*, and *final* are the routines used to initialize, update, and finalize a calculation context.

Table Management Routines

int **veriexec_file_add**(*struct lwp *l*, *prop_dictionary_t dict*)

Add a *Veriexec* entry for the file described by *dict*.

dict is expected to have the following:

Name	Type	Purpose

file	string	filename
entry-type	uint8	entry type flags (see `veriexec`(4))
fp-type	string	fingerprint hashing algorithm
fp	data	the fingerprint

int **veriexec_file_delete**(*struct lwp *l*, *struct vnode *vp*)
Remove *Veriexec* entry for *vp*.

int **veriexec_table_delete**(*struct lwp *l*, *struct mount *mp*)
Remove *Veriexec* table for mount-point *mp*.

int **veriexec_flush**(*struct lwp *l*)
Delete all *Veriexec* tables.

Hook Handlers

int **veriexec_openchk**(*struct lwp *l*, *struct vnode *vp*, *const char *path*, *int fmode*)
Called when a file is opened.

l is the LWP opening the file, *vp* is a vnode for the file being opened as returned from `namei`(9). If `NULL`, the file is being created. *path* is the pathname for the file (not necessarily a full path), and *fmode* are the mode bits with which the file was opened.

int **veriexec_renamechk**(*struct lwp *l*, *struct vnode *fromvp*, *const char *fromname*, *struct vnode *tovp*, *const char *toname*)
Called when a file is renamed.

fromvp and *fromname* are the vnode and filename of the file being renamed. *tovp* and *toname* are the vnode and filename of the target file. *l* is the LWP renaming the file.

Depending on the strict level, **veriexec** will either track changes appropriately or prevent the rename.

int **veriexec_removechk**(*struct lwp *l*, *struct vnode *vp*, *const char *name*)
Called when a file is removed.

vp is the vnode of the file being removed, and *name* is the filename. *l* is the LWP removing the file,

Depending on the strict level, **veriexec** will either clean-up after the file or prevent its removal.

int **veriexec_unmountchk**(*struct mount *mp*)
Checks if the current strict level allows *mp* to be unmounted.

Misc. Routines

int **veriexec_convert**(*struct vnode *vp*, *prop_dictionary_t rdict*)
Convert *Veriexec* entry for *vp* to human-readable `proplib`(3) dictionary, *rdict*, with the following elements:

Name	Type	Purpose
entry-type	uint8	entry type flags (see `veriexec`(4))
status	uint8	entry status (see below)
fp-type	string	fingerprint hashing algorithm
fp	data	the fingerprint

The "status" can be one of the following:

Status	Meaning
FINGERPRINT_NOTEVAL	not evaluated
FINGERPRINT_VALID	fingerprint match
FINGERPRINT_MISMATCH	fingerprint mismatch

If no entry was found, `ENOENT` is returned. Otherwise, zero.

int **veriexec_dump**(*struct lwp *l*, *prop_array_t rarray*)

Fill *rarray* with entries for all files monitored by *Veriexec* that have a filename associated with them.

Each element in *rarray* is a dictionary with the same elements as filled by **veriexec_convert**(), with an additional field, "file", containing the filename.

FILES

Path	Purpose
src/sys/dev/verified_exec.c	driver for userland communication
src/sys/sys/verified_exec.h	shared (userland/kernel) header file
src/sys/kern/kern_verifiedexec.c	subsystem code
src/sys/kern/vfs_syscalls.c	rename, remove, and unmount policies
src/sys/kern/vfs_vnops.c	regular file access policy

SEE ALSO

`proplib`(3), `sysctl`(3), `veriexec`(4), `security`(8), `sysctl`(8), `veriexecctl`(8), `veriexecgen`(8), `fileassoc`(9)

AUTHORS

Brett Lymn ⟨blymn@NetBSD.org⟩
Elad Efrat ⟨elad@NetBSD.org⟩

CAVEATS

There are two known issues with *Veriexec* that should be considered when using it.

Remote File-systems

There is an issue providing protection for files residing on mounts from remote hosts. Because access to the file-system does not necessarily go through **veriexec**, there is no way to track on-disk changes. While it is possible to minimize the effect by evaluating the file's fingerprint on each access without caching the result, a problem arises when a file is overwritten after its fingerprint has been evaluated and it is running on the local host.

An attacker could potentially overwrite the file contents in the remote host at that point, and force a flush on the local host, resulting in paging in of the files from the disk, introducing malicious code into a supposedly safe address space.

There is a fix for this issue, however due to dependencies on other work that is still in progress it has not been committed yet.

Layered File-systems

Due to VFS limitations, **veriexec** cannot track the same on-disk file across multiple layers of overlay file-systems. Therefore, you cannot expect changes to files on overlay mounts will be detected simply because the underlying mount is monitored by **veriexec**.

A workaround for this issue is listing all files, under all mounts, you want monitored in the signature file.

NAME

vfs — kernel interface to file systems

DESCRIPTION

The virtual file system, **vfs**, is the kernel interface to file systems. The interface specifies the calls for the kernel to access file systems. It also specifies the core functionality that a file system must provide to the kernel.

The focus of **vfs** activity is the *vnode* and is discussed in vnode(9). File system operations such as mounting and syncing are discussed in vfsops(9).

SEE ALSO

intro(9), vfsops(9), vnode(9), vnodeops(9)

NAME

vfs_hooks, **vfs_hooks_unmount** — VFS hooks interface

SYNOPSIS

```
#include <sys/param.h>
#include <sys/mount.h>

void
vfs_hooks_unmount(struct mount *mp);
```

DESCRIPTION

The VFS hooks interface provides a way for different kernel subsystems to attach custom functions to specific VFS operations. This enforces code separation by keeping the VFS's core sources uncluttered and makes all subsystem functionality reside in a single place. As an example, this interface is used by the NFS server code to automatically handle the exports list for each mount point.

Hooks are described by a *struct vfs_hooks* object, as seen below:

```
struct vfs_hooks {
        int     (*vh_unmount)(struct mount *);
};
```

For simplicity, each field is named after the VFS operation it refers to. The purpose of each member function, alongside some important notes, is shown below:

vh_unmount(*mp*)
: This hook is executed during the unmount process of a file system.

For more information about the purpose of each operation, see vfsops(9). Note that any of these fields may be a null pointer.

After the definition of a *struct vfs_hooks* object, the kernel has to add it to the *vfs_hooks* link set using the **VFS_HOOKS_ATTACH**(*struct vfs_hooks **) macro.

Please note that this interface is incomplete on purpose to keep it in its smallest possible size (i.e., do not provide a hook that is not used). If you feel the need to hook a routine to a VFS operation that is not yet supported by this interface, just add it to the files described in **CODE REFERENCES**.

FUNCTIONS

The following functions are provided to the VFS code to run the hooked functions:

vfs_hooks_unmount(*mp*)
: Runs all hooks for the VFS unmount operation. Given that these operations shall not fail, it returns *void*.

CODE REFERENCES

The VFS hooks interface is implemented within the files sys/kern/vfs_hooks.c and sys/sys/mount.h.

SEE ALSO

intro(9), vfs(9), vfsops(9)

HISTORY

The VFS hooks interface appeared in NetBSD 4.0.

NAME

vfsops, **VFS_MOUNT**, **VFS_START**, **VFS_UNMOUNT**, **VFS_ROOT**, **VFS_QUOTACTL**, **VFS_STATVFS**, **VFS_SYNC**, **VFS_VGET**, **VFS_FHTOVP**, **VFS_VPTOFH**, **VFS_SNAPSHOT**, **VFS_SUSPENDCTL** — kernel file system interface

SYNOPSIS

```
#include <sys/param.h>
#include <sys/mount.h>
#include <sys/vnode.h>

int
VFS_MOUNT(struct mount *mp, const char *path, void *data, size_t *dlen);

int
VFS_START(struct mount *mp, int flags);

int
VFS_UNMOUNT(struct mount *mp, int mntflags);

int
VFS_ROOT(struct mount *mp, struct vnode **vpp);

int
VFS_QUOTACTL(struct mount *mp, int cmds, uid_t uid, void *arg);

int
VFS_STATVFS(struct mount *mp, struct statvfs *sbp);

int
VFS_SYNC(struct mount *mp, int waitfor, kauth_cred_t cred);

int
VFS_VGET(struct mount *mp, ino_t ino, struct vnode **vpp);

int
VFS_FHTOVP(struct mount *mp, struct fid *fhp, struct vnode **vpp);

int
VFS_VPTOFH(struct vnode *vp, struct fid *fhp, size_t *fh_size);

int
VFS_SNAPSHOT(struct mount *mp, struct vnode *vp, struct timespec *ts);

int
VFS_SUSPENDCTL(struct mount *mp, int cmd);
```

DESCRIPTION

In a similar fashion to the vnode(9) interface, all operations that are done on a file system are conducted through a single interface that allows the system to carry out operations on a file system without knowing its construction or type.

All supported file systems in the kernel have an entry in the *vfs_list_initial* table. This table is generated by config(1) and is a NULL-terminated list of *vfsops* structures. The vfsops structure describes the operations that can be done to a specific file system type. The following table lists the elements of the vfsops vector, the corresponding invocation macro, and a description of the element.

Vector element	*Macro*	*Description*
int (*vfs_mount)()	VFS_MOUNT	Mount a file system
int (*vfs_start)()	VFS_START	Make operational

int (*vfs_unmount)()	VFS_UMOUNT	Unmount a file system
int (*vfs_root)()	VFS_ROOT	Get the file system root vnode
int (*vfs_quotactl)()	VFS_QUOTACTL	Query/modify space quotas
int (*vfs_statvfs)()	VFS_STATVFS	Get file system statistics
int (*vfs_sync)()	VFS_SYNC	Flush file system buffers
int (*vfs_vget)()	VFS_VGET	Get vnode from file id
int (*vfs_fhtovp)()	VFS_FHTOVP	NFS file handle to vnode lookup
int (*vfs_vptofh)()	VFS_VPTOFH	Vnode to NFS file handle lookup
void (*vfs_init)()	-	Initialize file system
void (*vfs_reinit)()	-	Reinitialize file system
void (*vfs_done)()	-	Cleanup unmounted file system
int (*vfs_mountroot)()	-	Mount the root file system
int (*vfs_snapshot)()	VFS_SNAPSHOT	Take a snapshot
int (*vfs_suspendctl)()	VFS_SUSPENDCTL	Suspend or resume

Some additional non-function members of the vfsops structure are the file system name *vfs_name* and a reference count *vfs_refcount*. It is not mandatory for a file system type to support a particular operation, but it must assign each member function pointer to a suitable function to do the minimum required of it. In most cases, such functions either do nothing or return an error value to the effect that it is not supported. *vfs_reinit*, *vfs_mountroot*, *vfs_fhtovp*, and *vfs_vptofh* may be `NULL`.

At system boot, each file system with an entry in *vfs_list_initial* is established and initialized. Each initialized file system is recorded by the kernel in the list *vfs_list* and the file system specific initialization function *vfs_init* in its vfsops vector is invoked. When the file system is no longer needed *vfs_done* is invoked to run file system specific cleanups and the file system is removed from the kernel list.

At system boot, the root file system is mounted by invoking the file system type specific *vfs_mountroot* function in the vfsops vector. All file systems that can be mounted as a root file system must define this function. It is responsible for initializing to list of mount structures for all future mounted file systems.

Kernel state which affects a specific file system type can be queried and modified using the `sysctl`(8) interface.

FUNCTIONS

VFS_MOUNT(*mp*, *path*, *data*, *dlen*)

Mount a file system specified by the mount structure *mp* on the mount point described by *path*. The argument *data* contains file system type specific data, while the argument *dlen* points to a location specifying the length of the data.

VFS_MOUNT() initializes the mount structure for the mounted file system. This structure records mount-specific information for the file system and records the list of vnodes associated with the file system. This function is invoked both to mount new file systems and to change the attributes of an existing file system. If the flag MNT_UPDATE is set in *mp->mnt_flag*, the file system should update its state. This can be used, for instance, to convert a read-only file system to read-write. The current attributes for a mounted file system can be fetched by specifying `MNT_GETARGS`. If neither `MNT_UPDATE` or `MNT_GETARGS` are specified, a new file system will attempted to be mounted.

VFS_START(*mp*, *flags*)

Make the file system specified by the mount structure *mp* operational. The argument *flags* is a set of flags for controlling the operation of **VFS_START**(). This function is invoked after **VFS_MOUNT**() and before the first access to the file system.

VFS_UNMOUNT(*mp*, *mntflags*)
Unmount a file system specified by the mount structure *mp*. **VFS_UNMOUNT**() performs any file system type specific operations required before the file system is unmounted, such are flushing buffers. If MNT_FORCE is specified in the flags *mntflags* then open files are forcibly closed. The function also deallocates space associated with data structure that were allocated for the file system when it was mounted.

VFS_ROOT(*mp*, *vpp*)
Get the root vnode of the file system specified by the mount structure *mp*. The vnode is returned in the address given by *vpp*. This function is used by the pathname translation algorithms when a vnode that has been covered by a mounted file system is encountered. While resolving the pathname, the pathname translation algorithm will have to go through the directory tree in the file system associated with that mount point and therefore requires the root vnode of the file system.

VFS_QUOTACTL(*mp*, *cmds*, *uid*, *arg*)
Query/modify user space quotas for the file system specified by the mount structure *mp*. The argument specifies the control command to perform. The userid is specified in *id* and *arg* allows command-specific data to be returned to the system call interface. **VFS_QUOTACTL**() is the file system type specific implementation of the `quotactl`(2) system call.

VFS_STATVFS(*mp*, *sbp*)
Get file system statistics for the file system specified by the mount structure *mp*. A statvfs structure filled with the statistics is returned in *sbp*. **VFS_STATVFS**() is the file system type specific implementation of the `statvfs`(2) and `fstatvfs`(2) system calls.

VFS_SYNC(*mp*, *waitfor*, *cred*)
Flush file system I/O buffers for the file system specified by the mount structure *mp*. The *waitfor* argument indicates whether a partial flush or complete flush should be performed. The argument *cred* specifies the calling credentials. **VFS_SYNC**() does not provide any return value since the operation can never fail.

VFS_VGET(*mp*, *ino*, *vpp*)
Get vnode for a file system type specific file id *ino* for the file system specified by the mount structure *mp*. The vnode is returned in the address specified *vpp*. The function is optional for file systems which have a unique id number for every file in the file system. It is used internally by the UFS file system and also by the NFSv3 server to implement the READDIRPLUS NFS call. If the file system does not support this function, it should return `EOPNOTSUPP`.

VFS_FHTOVP(*mp*, *fhp*, *vpp*)
Get the vnode for the file handle *fhp* in the file system specified by the mount structure *mp*. The locked vnode is returned in *vpp*.

When exporting, the call to **VFS_FHTOVP**() should follow a call to **netexport_check**(), which checks if the file is accessible to the client.

If file handles are not supported by the file system, this function must return `EOPNOTSUPP`.

VFS_VPTOFH(*vp*, *fhp*, *fh_size*)
Get a file handle for the vnode specified by *vp*. The file handle is returned in *fhp*. The contents of the file handle are defined by the file system and are not examined by any other subsystems. It should contain enough information to uniquely identify a file within the file system as well as noticing when a file has been removed and the file system resources have been recycled for a new file.

The parameter *fh_size* points to the container size for the file handle. This parameter should be updated to the size of the finished file handle. Note that it is legal to call this function with *fhp* set to `NULL` in case *fh_size* is zero. In case *fh_size* indicates a storage space too

small, the storage space required for the file handle corresponding to *vp* should be filled in and E2BIG should be returned.

If file handles are not supported by the file system, this function must return EOPNOTSUPP.

VFS_SNAPSHOT(*mp*, *vp*, *ts*)
Take a snapshot of the file system specified by the mount structure *mp* and make it accessible through the locked vnode *vp*. If *ts* is not NULL it will receive the time this snapshot was taken. If the file system does not support this function, it should return EOPNOTSUPP.

VFS_SUSPENDCTL(*mp*, *cmd*)
Suspend or resume all operations on this file system. *cmd* is either SUSPEND_SUSPEND to suspend or SUSPEND_RESUME to resume operations. If the file system does not support this function, it should return EOPNOTSUPP.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the vfs operations can be found. All pathnames are relative to /usr/src.

The vfs operations are implemented within the files sys/kern/vfs_subr.c, sys/kern/vfs_subr2.c and sys/kern/vfs_init.c.

SEE ALSO

intro(9), namei(9), vfs(9), vfssubr(9), vnode(9), vnodeops(9)

HISTORY

The vfs operations vector, its functions and the corresponding macros appeared in 4.3BSD.

NAME

vfssubr, **vfs_getnewfsid**, **vfs_getvfs**, **vfs_export**, **vfs_showexport**, **vfs_export_lookup**, **vfs_setpublicfs**, **vfs_mountedon**, **vfs_mountroot**, **vfs_unmountall**, **vfs_busy**, **vfs_unbusy**, **vfs_rootmountalloc**, **vfs_shutdown**, **vfs_attach**, **vfs_detach**, **vfs_reinit**, **vfs_getopsbyname**, **vfs_suspend**, **vfs_resume** — high-level interface to kernel file system interface

SYNOPSIS

```
#include <sys/param.h>
#include <sys/mount.h>
#include <sys/vnode.h>

void
vfs_getnewfsid(struct mount *mp);

struct mount *
vfs_getvfs(fsid_t *fsid);

int
vfs_export_lookup(struct mount *mp, struct netexport *nep,
    struct export_args *argp);

int
vfs_setpublicfs(struct mount *mp, struct netexport *nep,
    struct export_args *argp);

int
vfs_mountedon(struct vnode *vp);

int
vfs_mountroot(void);

void
vfs_unmountall(struct lwp *l);

int
vfs_busy(struct mount *mp, int flags, struct simplelock *interlkp);

void
vfs_unbusy(struct mount *mp);

int
vfs_rootmountalloc(char *fstypename, char *devname, struct mount **mpp);

void
vfs_shutdown(void);

int
vfs_attach(struct vfsops *vfs);

int
vfs_detach(struct vfsops *vfs);

void
vfs_reinit(void);

struct vfsops *
vfs_getopsbyname(const char *name);
```

int
vfs_suspend(*struct mount *mp*, *int nowait*);

void
vfs_resume(*struct mount *mp*);

DESCRIPTION

The high-level functions described in this page are the interface to the kernel file system interface (VFS).

FUNCTIONS

vfs_getnewfsid(*mp*)
Get a new unique file system id type for the file system specified by the mount structure *mp*. The file system id type is stored in *mp->mnt_stat.f_fsidx*.

vfs_getvfs(*fsid*)
Lookup a mount point with the file system identifier *fsid*.

vfs_export_lookup(*mp*, *nep*, *argp*)
Check client permission on the exportable file system specified by the mount structure *mp*. The argument *nam* is the address of the networked client. This function is used by file system type specific functions to verify that the client can access the file system.

vfs_setpublicfs(*mp*, *nep*, *argp*)
Set the publicly exported file system specified by the mount structure *mp*.

vfs_mountedon(*vp*)
Check to see if a file system is mounted on a block device specified by the vnode *vp*.

vfs_mountroot(*void*)
Mount the root file system.

vfs_unmountall(*l*)
Unmount all file systems.

vfs_busy(*mp*, *flags*, *interlkp*)
Mark the mount point specified by *mp* as busy. This function is used to synchronize access and to delay unmounting. The interlock specified by argument *interlkp* is not released on failure.

vfs_unbusy(*mp*)
Free the busy file system specified by the mount structure *mp*.

vfs_rootmountalloc(*fstypename*, *devname*, *mpp*)
Lookup a file system type specified by the name *fstypename* and if found allocate and initialise a mount structure for it. The allocated mount structure is returned in the address specified by *mpp*. The device the root file system was mounted from is specified by the argument *devname* and is recorded in the new mount structure.

vfs_shutdown()
Sync and unmount all file systems before shutting down. Invoked by cpu_reboot(9).

vfs_attach(*vfs*)
Establish file system *vfs* and initialise it.

vfs_detach(*vfs*)
Remove file system *vfs* from the kernel.

vfs_reinit(*void*)
Reinitialises all file systems within the kernel through file system-specific vfs operation (see vfsops(9)).

vfs_getopsbyname(*name*)
Given a file system name specified by *name*, look up the vfs operations for that file system (see vfsops(9)), or return NULL if file system isn't present in the kernel.

vfs_suspend(*mp*, *nowait*)
Request a mounted file system to suspend all operations. All new operations to the file system are stopped. After all operations in progress have completed, the file system is synced to disk and the function returns. If a file system suspension is currently in progress and *nowait* is set EWOULDBLOCK is returned. If the operation is successful, zero is returned, otherwise an appropriate error code is returned.

vfs_resume(*mp*)
Request a mounted file system to resume operations.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the vfs operations can be found. All pathnames are relative to /usr/src.

The vfs interface functions are implemented within the files sys/kern/vfs_subr.c, sys/kern/vfs_subr2.c, and sys/kern/vfs_init.c.

SEE ALSO

intro(9), namei(9), vfs(9), vfsops(9), vnode(9), vnodeops(9)

NAME

VME, **vme_probe**, **vme_space_map**, **vme_space_unmap**, **vme_intr_map**, **vme_intr_establish**, **vme_intr_disestablish**, **vme_intr_evcnt**, **vme_dmamap_create**, **vme_dmamap_destroy**, **vme_dmamem_alloc**, **vme_dmamem_free**, **vme_space_alloc**, **vme_space_free**, **vme_space_get** — Versa Module Euroboard bus

SYNOPSIS

```
#include <machine/bus.h>
#include <dev/vme/vmereg.h>
#include <dev/vme/vmevar.h>

int
vme_probe(void *vc, vme_addr_t vmeaddr, vme_size_t len, vme_am_t am,
    vme_datasize_t datasize, int (*callback)(), void *arg);

int
vme_space_map(void *vc, vme_addr_t vmeaddr, vme_size_t len, vme_am_t am,
    vme_datasize_t datasize, vme_swap_t swap, bus_space_tag_t *tag,
    bus_space_handle_t *handle, vme_mapresc_t *resc);

void
vme_space_unmap(void *vc, vme_mapresc_t resc);

int
vme_intr_map(void *vc, int level, int vector, vme_intr_handle_t *handlep);

void *
vme_intr_establish(void *vc, vme_intr_handle_t handle, int prio,
    int (*func)(void *), void *arg);

void
vme_intr_disestablish(void *vc, void *cookie);

const struct evcnt *
vme_intr_evcnt(void *vc, vme_intr_handle_t handle);

int
vme_dmamap_create(void *vc, vme_size_t size, vme_am_t am,
    vme_datasize_t datasize, vme_swap_t swap, int nsegs, vme_size_t segsz,
    vme_addr_t bound, int flags, bus_dmamap_t *map);

void
vme_dmamap_destroy(void *vc, bus_dmamap_t map);

int
vme_dmamem_alloc(void *vc, vme_size_t size, vme_am_t am,
    vme_datasize_t datasize, vme_swap_t swap, bus_dma_segment_t *segs,
    int nsegs, int *rsegs, int flags);

void
vme_dmamem_free(void *vc, bus_dma_segment_t *segs, int nsegs);

int
vme_space_alloc(struct vmebus_softc *tag, vme_addr_t addr, vme_size_t size,
    vme_am_t ams);

void
vme_space_free(void *vc, vme_addr_t addr, vme_size_t size, vme_am_t ams);
```

int
vme_space_get(*void *vc*, *vme_size_t size*, *vme_am_t ams*, *u_long align*, *vme_addr_t *addr*);

DESCRIPTION

The **VME** bus provides support for VME devices. The VME bus is a high-performance backplane bus for use in computer systems. It is based on the VMEbus specification initially released by the VMEbus International Trade Association (VITA) in August of 1982. It has since undergone IEC and IEEE standardisation.

The VME bus supports 8, 16, and 32-bit transfers over non-multiplexed 32-bit data and address paths. The latest revisions allow 64-bit, multiplexed transfers. It supports asynchronous, fully handshaken transfers at speeds up to 80 MB/sec. It has a master-slave architecture, encouraging multiprocessing and supports up to seven interrupt levels.

DATA TYPES

Drivers attached to the **VME** bus will make use of the following data types:

vme_chipset_tag_t
: An opaque type identifying the bus controller.

vme_addr_t
: Addresses on the bus.

vme_am_t
: Address modifiers. Valid values are VME_AM_A32, VME_AM_A16, VME_AM_A24, VME_AM_USERDEF (user/vendor definable), VME_AM_MBO, VME_AM_SUPER, VME_AM_USER, VME_AM_DATA, VME_AM_PRG, VME_AM_BLT32 and VME_AM_BLT64.

vme_datasize_t
: The datasize of the address space. Valid values are VME_D8, VME_D16, and VME_D32.

vme_mapresc_t
: Generic placeholder for any resources needed for a mapping.

vme_intr_handle_t
: An opaque type describing an interrupt mapping.

vme_swap_t
: Hardware swap capabilities for controlling data endianness. Valid values have not been specified yet.

struct vme_range
: A structure used to describe an address range on the VME bus. It contains the following members:

```
vme_addr_t offset;
vme_size_t size;
vme_am_t am;
```

struct vme_attach_args
: A structure used to inform the driver of the device properties. It contains the following members:

```
vme_chipset_tag_t va_vct;
bus_dma_tag_t va_bdt;
int ivector;
int ilevel;
int numcfranges;
```

```
struct vme_range r[VME_MAXCFRANGES];
```

FUNCTIONS

vme_probe(*vc*, *vmeaddr*, *len*, *am*, *datasize*, *callback*, *arg*)
Probes the VME space managed by controller *vc* at address *vmeaddr*, length *len*, with address modifiers *am* and datasize *datasize* for a device. If a VME device is found, the function *callback()* (if it is not NULL) is called to perform device-specific identification. *callback()* is called with the argument *arg*, and the bus-space tag and bus-space handle for accessing the VME space mapping and should return a nonzero positive integer for a positive device match.

vme_space_map(*vc*, *vmeaddr*, *len*, *am*, *datasize*, *swap*, *tag*, *handle*, *resc*)
Maps the VME space managed by controller *vc* at address *vmeaddr*, length *len*, with address modifiers *am*, datasize *datasize* and endianness *swap* for a device. If the mapping is successful *tag* contains the bus-space tag and *handle* contains the bus-space handle for accessing the VME space mapping. *resc* contains the resources for the mappings. **vme_space_map**() returns 0 on success, and nonzero on error.

vme_space_unmap(*vc*, *resc*)
Unmaps the VME space mapping managed by controller *vc* and resources *resc*.

vme_intr_map(*vc*, *level*, *vector*, *handlep*)
Sets *handlep* to a machine-dependent value which identifies a particular interrupt source at level *level* and vector *vector* on the controller *vc*. **vme_intr_map**() returns zero on success, and nonzero on failure.

vme_intr_establish(*vc*, *handle*, *prio*, *func*, *arg*)
Establishes the interrupt handler *handlep*. When the device interrupts, *func()* will be called with a single argument *arg* and will run at the interrupt priority level *prio*. The return value of **vme_intr_establish**() may be saved and passed to **vme_intr_disestablish**().

vme_intr_disestablish(*vc*, *cookie*)
Disables the interrupt handler when the driver is no longer interested in interrupts from the device. *cookie* is the value returned by **vme_intr_establish**().

vme_intr_evcnt(*vc*, *handle*)
Increment the interrupt event counter for the interrupt specified by *handle*.

vme_dmamap_create(*vc*, *size*, *am*, *datasize*, *swap*, *nsegs*, *segsz*, *bound*, *flags*, *map*)
Allocates a DMA handle and initializes it according to the parameters provided. The VME-specific parameters describe the address-space modifiers *am*, datasize *datasize*, and endianness *swap*. The remaining parameters are described in bus_dma(9).

vme_dmamap_destroy(*vc*, *map*)
Frees all resources associated with a given DMA handle. The parameters are described in bus_dma(9).

vme_dmamem_alloc(*vc*, *size*, *am*, *datasize*, *swap*, *segs*, *nsegs*, *rsegs*, *flags*)
Allocates memory that is "DMA safe" for the VME bus managed by controller *vc*. The VME-specific parameters describe the address-space modifiers *am*, datasize *datasize*, and endianness *swap*. The remaining parameters are described in bus_dma(9).

vme_dmamem_free(*vc*, *segs*, *nsegs*)
Frees memory previously allocated by **vme_dmamem_alloc**() for the VME space managed by controller *vc*.

vme_space_alloc(*tag*, *addr*, *size*, *ams*)
Allocate VME space for the bus-space *tag* at address *addr* of size *size* and address-space modifiers *ams*. **vme_space_alloc**() returns EINVAL on invalid inputs.

vme_space_free(*vc*, *addr*, *size*, *ams*)
Deallocate VME space for the bus-space *tag* at address *addr* of size *size* and address-space modifiers *ams*.

vme_space_get(*vc*, *size*, *ams*, *align*, *addr*)
Returns EINVAL on invalid inputs.

AUTOCONFIGURATION

The VME bus is an indirect-connection bus. During autoconfiguration each driver is required to probe the bus for the presence of a device. A VME driver will receive a pointer to a *struct vme_attach_args* hinting at "locations" (address ranges) on the VME bus where the device may be located. The driver should check the number of address ranges, allocate the address space of these ranges using **vme_space_alloc**(), and probe the address space for the device using **vme_probe**().

During driver attach the driver should also map the address ranges using **vme_space_map**(). The interrupt locators in *struct vme_attach_args* are used by **vme_intr_map**() and **vme_intr_establish**().

DMA SUPPORT

Extensive DMA facilities are provided.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent **VME** subsystem can be found. All pathnames are relative to `/usr/src`.

The **VME** subsystem itself is implemented within the file `sys/dev/vme/vme.c`.

SEE ALSO

`vme`(4), `autoconf`(9), `bus_dma`(9), `bus_space`(9), `driver`(9)

HISTORY

The machine-independent VME subsystem appeared in NetBSD 1.5.

BUGS

This page is incomplete.

NAME

vmem — virtual memory allocator

DESCRIPTION

The **vmem** is a general purpose resource allocator. Despite its name, it can be used for arbitrary resources other than virtual memory.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing the **vmem** subsystem can be found. All pathnames are relative to `/usr/src`.

The **vmem** subsystem is implemented within the file `sys/kern/subr_vmem.c`.

SEE ALSO

`intro`(9), `memoryallocators`(9), `vmem_alloc`(9), `vmem_create`(9), `vmem_destroy`(9), `vmem_free`(9), `vmem_xalloc`(9), `vmem_xfree`(9)

Jeff Bonwick and Jonathan Adams, "Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources", *2001 USENIX Annual Technical Conference*, 2001.

AUTHORS

This implementation of **vmem** was written by YAMAMOTO Takashi.

NAME

vmem_alloc — Allocate resource from arena

SYNOPSIS

```
#include <sys/vmem.h>

vmem_addr_t
vmem_alloc(vmem_t *vm, vmem_size_t size, vm_flag_t flags);
```

DESCRIPTION

vmem_alloc() allocates a resource from the arena.

vm The arena which we allocate from.

size Specify the size of the allocation.

flags A bitwise OR of an allocation strategy and a sleep flag.

The allocation strategy is one of:

VM_BESTFIT Prefer space efficiency.

VM_INSTANTFIT Prefer performance.

The sleep flag should be one of:

VM_SLEEP Can sleep until enough resources are available.

VM_NOSLEEP Don't sleep. Immediately return VMEM_ADDR_NULL if there are not enough resources available.

RETURN VALUES

On success, **vmem_alloc**() returns an allocated vmem_addr_t. Otherwise, it returns VMEM_ADDR_NULL.

SEE ALSO

intro(9), vmem(9)

NAME

vmem_create — create a vmem arena

SYNOPSIS

```
#include <sys/vmem.h>

vmem_t
vmem_create(const char *name, vmem_addr_t base, vmem_size_t size,
    vmem_size_t quantum,
    vmem_addr_t (*allocfn)(vmem_t *, vmem_size_t, vmem_size_t *, vm_flag_t),
    void (*freefn)(vmem_t *, vmem_addr_t, vmem_size_t), vmem_t *source,
    vmem_size_t qcache_max, vm_flag_t flags);
```

DESCRIPTION

vmem_create() creates a new vmem arena.

name The string to describe the vmem.

base The start address of the initial span. It can be VMEM_ADDR_NULL if no initial span is required.

size The size of the initial span.

quantum The smallest unit of allocation.

allocfn The callback function used to import spans from the backend arena.

freefn The callback function used to free spans to the backend arena.

source The backend arena.

qcache_max The largest size of allocations which can be served by quantum cache. It is merely a hint and can be ignored.

flags Either of:

VM_SLEEP Can sleep until enough resources are available.

VM_NOSLEEP Don't sleep. Immediately return NULL if there are not enough resources available.

RETURN VALUES

vmem_create() return a pointer to the newly allocated vmem_t. Otherwise, it returns NULL.

SEE ALSO

intro(9), vmem(9)

NAME

vmem_create — destroy a vmem arena

SYNOPSIS

```
#include <sys/vmem.h>

void
vmem_destroy(vmem_t *vm);
```

DESCRIPTION

vmem_destroy() destroys a vmem arena.

vm The vmem arena being destroyed. The caller should ensure that no one will use it anymore.

SEE ALSO

intro(9), vmem(9)

NAME

vmem_free — free resource to arena

SYNOPSIS

```
#include <sys/vmem.h>

void
vmem_free(vmem_t *vm, vmem_addr_t addr, vmem_size_t size);
```

DESCRIPTION

vmem_free() frees resource allocated by *vmem_alloc* to the arena.

vm The arena which we free to.

addr The resource being freed. It must be the one returned by **vmem_alloc**(). Notably, it must not be the one from **vmem_xalloc**(). Otherwise, the behaviour is undefined.

size The size of the resource being freed. It must be the same as the *size* argument used for **vmem_alloc**().

SEE ALSO

intro(9), vmem(9)

NAME

vmem_xalloc — Allocate resource from arena

SYNOPSIS

```
#include <sys/vmem.h>

vmem_addr_t
vmem_xalloc(vmem_t *vm, vmem_size_t size, vmem_size_t align,
    vmem_size_t phase, vmem_size_t nocross, vmem_addr_t minaddr,
    vmem_addr_t maxaddr, vm_flag_t flags);
```

DESCRIPTION

vmem_xalloc() allocates a resource from the arena.

vm The arena which we allocate from.

size Specify the size of the allocation.

align If zero, don't care about the alignment of the allocation. Otherwise, request a resource segment starting at offset *phase* from an *align* aligned boundary.

phase See the above description of *align*. If *align* is zero, *phase* should be zero. Otherwise, *phase* should be smaller than *align*.

nocross Request a resource which doesn't cross *nocross* aligned boundary.

minaddr If non-zero, specify the minimum address which can be allocated.

maxaddr If non-zero, specify the maximum address + 1 which can be allocated.

flags A bitwise OR of an allocation strategy and a sleep flag.

The allocation strategy is one of:

VM_BESTFIT Prefer space efficiency.

VM_INSTANTFIT Prefer performance.

The sleep flag should be one of:

VM_SLEEP Can sleep until enough resources are available.

VM_NOSLEEP Don't sleep. Immediately return VMEM_ADDR_NULL if there are not enough resources available.

RETURN VALUES

On success, **vmem_xalloc**() returns an allocated vmem_addr_t. Otherwise, it returns VMEM_ADDR_NULL.

SEE ALSO

intro(9), vmem(9)

NAME

vmem_xfree — free resource to arena

SYNOPSIS

```
#include <sys/vmem.h>

void
vmem_xfree(vmem_t *vm, vmem_addr_t addr, vmem_size_t size);
```

DESCRIPTION

vmem_xfree() frees resource allocated by *vmem_xalloc* to the arena.

vm The arena which we free to.

addr The resource being freed. It must be the one returned by **vmem_xalloc**(). Notably, it must not be the one from **vmem_alloc**(). Otherwise, the behaviour is undefined.

size The size of the resource being freed. It must be the same as the *size* argument used for **vmem_xalloc**().

SEE ALSO

intro(9), vmem(9)

NAME

vnfileops, **vn_closefile**, **vn_fcntl**, **vn_ioctl**, **vn_read**, **vn_poll**, **vn_statfile**, **vn_write** — vnode file descriptor operations

SYNOPSIS

```
#include <sys/param.h>
#include <sys/file.h>
#include <sys/vnode.h>

int
vn_closefile(file_t *fp);

int
vn_fcntl(file_t *fp, u_int com, void *data);

int
vn_ioctl(file_t *fp, u_long com, void *data);

int
vn_read(file_t *fp, off_t *offset, struct uio *uio, kauth_cred_t cred,
    int flags);

int
vn_poll(file_t *fp, int events);

int
vn_statfile(file_t *fp, struct stat *sb);

int
vn_write(file_t *fp, off_t *offset, struct uio *uio, kauth_cred_t cred,
    int flags);
```

DESCRIPTION

The functions described in this page are the vnode-specific file descriptor operations. They should only be accessed through the opaque function pointers in the file entries (see file(9)). They are described here only for completeness.

FUNCTIONS

vn_closefile(*fp*, *l*)

Common code for a file table vnode close operation. The file is described by *fp* and *l* is the calling lwp. **vn_closefile**() simply calls vn_close(9) with the appropriate arguments.

vn_fcntl(*fp*, *com*, *data*, *l*)

Common code for a file table vnode fcntl(2) operation. The file is specified by *fp*. The argument *l* is the calling lwp. **vn_fcntl**() simply locks the vnode and invokes the vnode operation VOP_FCNTL(9) with the command *com* and buffer *data*. The vnode is unlocked on return. If the operation is successful zero is returned, otherwise an appropriate error is returned.

vn_ioctl(*fp*, *com*, *data*, *l*)

Common code for a file table vnode ioctl operation. The file is specified by *fp*. The argument *l* is the calling lwp **vn_ioctl**() simply locks the vnode and invokes the vnode operation VOP_IOCTL(9) with the command *com* and buffer *data*. The vnode is unlocked on return. If the operation is successful zero is returned, otherwise an appropriate error is returned.

vn_read(*fp*, *offset*, *uio*, *cred*, *flags*)

Common code for a file table vnode read. The argument *fp* is the file structure, The argument *offset* is the offset into the file. The argument *uio* is the uio structure describing the memory

to read into. The caller's credentials are specified in *cred*. The *flags* argument can define FOF_UPDATE_OFFSET to update the read position in the file. If the operation is successful zero is returned, otherwise an appropriate error is returned.

vn_poll(*fp*, *events*, *l*)
Common code for a file table vnode poll operation. **vn_poll**() simply calls VOP_POLL(9) with the events *events* and the calling lwp *l*. The function returns a bitmask of available events.

vn_statfile(*fp*, *sb*, *l*)
Common code for a stat operation. The file descriptor is specified by the argument *fp* and *sb* is the buffer to return the stat information. The argument *l* is the calling lwp. **vn_statfile**() basically calls the vnode operation VOP_GETATTR(9) and transfer the contents of a vattr structure into a struct stat. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

vn_write(*fp*, *offset*, *uio*, *cred*, *flags*)
Common code for a file table vnode write. The argument *fp* is the file structure, The argument *offset* is the offset into the file. The argument *uio* is the uio structure describing the memory to read from. The caller's credentials are specified in *cred*. The *flags* argument can define FOF_UPDATE_OFFSET to update the read position in the file. If the operation is successful zero is returned, otherwise an appropriate error is returned.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the vnode framework can be found. All pathnames are relative to /usr/src.

The high-level convenience functions are implemented within the file sys/kern/vfs_vnops.c.

SEE ALSO

file(9), intro(9), vnode(9), vnodeops(9), vnsubr(9)

NAME

vnode, **vcount**, **vref**, **VREF**, **vrele**, **vget**, **vput**, **vhold**, **VHOLD**, **holdrele**, **HOLDRELE**, **getnewvnode**, **ungetnewvnode**, **vrecycle**, **vgone**, **vgonel**, **vflush**, **vaccess**, **checkalias**, **bdevvp**, **cdevvp**, **vfinddev**, **vdevgone**, **vwakeup**, **vflushbuf**, **vinvalbuf**, **vtruncbuf**, **vprint** — kernel representation of a file or directory

SYNOPSIS

```
#include <sys/param.h>
#include <sys/vnode.h>

int
vcount(struct vnode *vp);

void
vref(struct vnode *vp);

void
VREF(struct vnode *vp);

void
vrele(struct vnode *vp);

int
vget(struct vnode *vp, int lockflag);

void
vput(struct vnode *vp);

void
vhold(struct vnode *vp);

void
VHOLD(struct vnode *vp);

void
holdrele(struct vnode *vp);

void
HOLDRELE(struct vnode *vp);

int
getnewvnode(enum vtagtype tag, struct mount *mp, int (**vops)(void *),
    struct vnode **vpp);

void
ungetnewvnode(struct vnode *vp);

int
vrecycle(struct vnode *vp, struct simplelock *inter_lkp, struct lwp *l);

void
vgone(struct vnode *vp);

void
vgonel(struct vnode *vp, struct lwp *l);

int
vflush(struct mount *mp, struct vnode *skipvp, int flags);
```

int
vaccess(*enum vtype type*, *mode_t file_mode*, *uid_t uid*, *gid_t gid*, *mode_t acc_mode*, *kauth_cred_t cred*);

*struct vnode **
checkalias(*struct vnode *vp*, *dev_t nvp_rdev*, *struct mount *mp*);

int
bdevvp(*dev_t dev*, *struct vnode **vpp*);

int
cdevvp(*dev_t dev*, *struct vnode **vpp*);

int
vfinddev(*dev_t dev*, *enum vtype*, *struct vnode **vpp*);

void
vdevgone(*int maj*, *int minl*, *int minh*, *enum vtype type*);

void
vwakeup(*struct buf *bp*);

void
vflushbuf(*struct vnode *vp*, *int sync*);

int
vinvalbuf(*struct vnode *vp*, *int flags*, *kauth_cred_t cred*, *struct lwp *l*, *int slpflag*, *int slptimeo*);

int
vtruncbuf(*struct vnode *vp*, *daddr_t lbn*, *int slpflag*, *int slptimeo*);

void
vprint(*const char *label*, *struct vnode *vp*);

DESCRIPTION

The vnode is the focus of all file activity in NetBSD. There is a unique vnode allocated for each active file, directory, mounted-on file, fifo, domain socket, symbolic link and device. The kernel has no concept of a file's underlying structure and so it relies on the information stored in the vnode to describe the file. Thus, the vnode associated with a file holds all the administration information pertaining to it.

When a process requests an operation on a file, the `vfs`(9) interface passes control to a file system type dependent function to carry out the operation. If the file system type dependent function finds that a vnode representing the file is not in main memory, it dynamically allocates a new vnode from the system main memory pool. Once allocated, the vnode is attached to the data structure pointer associated with the cause of the vnode allocation and it remains resident in the main memory until the system decides that it is no longer needed and can be recycled.

The vnode has the following structure:

```
struct vnode {
        struct uvm_object v_uobj;               /* uvm object */
#define v_usecount      v_uobj.uo_refs
#define v_interlock     v_uobj.vmobjlock
        voff_t          v_size;                 /* size of file */
        int             v_flag;                 /* flags */
        int             v_numoutput;            /* num pending writes */
        long            v_writecount;           /* ref count of writers */
```

```
        long            v_holdcnt;              /* page & buffer refs */
        struct mount    *v_mount;               /* ptr to vfs we are in */
        int             (**v_op)(void *);       /* vnode ops vector */
        TAILQ_ENTRY(vnode) v_freelist;          /* vnode freelist */
        LIST_ENTRY(vnode) v_mntvnodes;          /* vnodes for mount pt */
        struct buflists v_cleanblkhd;           /* clean blocklist head */
        struct buflists v_dirtyblkhd;           /* dirty blocklist head */
        LIST_ENTRY(vnode) v_synclist;           /* dirty vnodes */
        LIST_HEAD(, namecache) v_dnclist;       /* namecaches for children */
        LIST_HEAD(, namecache) v_nclist;        /* namecaches for our parent */
        union {
                struct mount    *vu_mountedhere;/* ptr to mounted vfs */
                struct socket   *vu_socket;     /* unix ipc (VSOCK) */
                struct specinfo *vu_specinfo;   /* device (VCHR, VBLK) */
                struct fifoinfo *vu_fifoinfo;   /* fifo (VFIFO) */
        } v_un;
#define v_mountedhere   v_un.vu_mountedhere
#define v_socket        v_un.vu_socket
#define v_specinfo      v_un.vu_specinfo
#define v_fifoinfo      v_un.vu_fifoinfo
        struct nqlease  *v_lease;               /* Soft ref to lease */
        enum vtype      v_type;                 /* vnode type */
        enum vtagtype   v_tag;                  /* underlying data type */
        struct lock     v_lock;                 /* lock for this vnode */
        struct lock     *v_vnlock;              /* ptr to vnode lock */
        void            *v_data;                /* private data for fs */
        struct klist    v_klist;                /* knotes attached to vnode */
};
```

Most members of the vnode structure should be treated as opaque and only manipulated using the proper functions. There are some rather common exceptions detailed throughout this page.

Files and file systems are inextricably linked with the virtual memory system and *v_uobj* contains the data maintained by the virtual memory system. For compatibility with code written before the integration of uvm(9) into NetBSD, C-preprocessor directives are used to alias the members of *v_uobj*.

Vnode flags are recorded by *v_flag*. Valid flags are:

VROOT This vnode is the root of its file system.

VTEXT This vnode is a pure text prototype.

VSYSTEM This vnode is being used by the kernel; only used to skip quota files in **vflush**().

VISTTY This vnode represents a tty; used when reading dead vnodes.

VEXECMAP This vnode has executable mappings.

VWRITEMAP
This vnode might have PROT_WRITE user mappings.

VWRITEMAPDIRTY
This vnode might have dirty pages due to VWRITEMAP

VLOCKSWORK
This vnode's file system supports locking.

VXLOCK This vnode is currently locked to change underlying type.

VXWANT A process is waiting for this vnode.

VBWAIT Waiting for output associated with this vnode to complete.
VALIASED This vnode has an alias.
VDIROP This vnode is involved in a directory operation. This flag is used exclusively by LFS.
VLAYER This vnode is on a layered file system.
VONWORKLST
This vnode is on syncer work-list.
VFREEING This vnode is being freed.
VMAPPED This vnode might have user mappings.

The VXLOCK flag is used to prevent multiple processes from entering the vnode reclamation code. It is also used as a flag to indicate that reclamation is in progress. The VXWANT flag is set by threads that wish to be awakened when reclamation is finished. Before *v_flag* can be modified, the *v_interlock* simplelock must be acquired. See `lock`(9) for details on the kernel locking API.

Each vnode has three reference counts: *v_usecount*, *v_writecount* and *v_holdcnt*. The first is the number of active references within the kernel to the vnode. This count is maintained by **`vref`**(), **`vrele`**(), and **`vput`**(). The second is the number of active references within the kernel to the vnode performing write access to the file. It is maintained by the `open`(2) and `close`(2) system calls. The third is the number of references within the kernel requiring the vnode to remain active and not be recycled. This count is maintained by **`vhold`**() and **`holdrele`**(). When both the *v_usecount* and *v_holdcnt* reach zero, the vnode is recycled to the freelist and may be reused for another file. The transition to and from the freelist is handled by **`getnewvnode`**(), **`ungetnewvnode`**() and **`vrecycle`**(). Access to *v_usecount*, *v_writecount* and *v_holdcnt* is also protected by the *v_interlock* simplelock.

The number of pending synchronous and asynchronous writes on the vnode are recorded in *v_numoutput*. It is used by `fsync`(2) to wait for all writes to complete before returning to the user. Its value must only be modified at splbio (see `spl`(9)). It does not track the number of dirty buffers attached to the vnode.

v_dnclist and *v_nclist* are used by `namecache`(9) to maintain the list of associated entries so that `cache_purge`(9) can purge them.

The link to the file system which owns the vnode is recorded by *v_mount*. See `vfsops`(9) for further information of file system mount status.

The *v_op* pointer points to its vnode operations vector. This vector describes what operations can be done to the file associated with the vnode. The system maintains one vnode operations vector for each file system type configured into the kernel. The vnode operations vector contains a pointer to a function for each operation supported by the file system. See `vnodeops`(9) for a description of vnode operations.

When not in use, vnodes are kept on the freelist through *v_freelist*. The vnodes still reference valid files but may be reused to refer to a new file at any time. When a valid vnode which is on the freelist is used again, the user must call **`vget`**() to increment the reference count and retrieve it from the freelist. When a user wants a new vnode for another file, **`getnewvnode`**() is invoked to remove a vnode from the freelist and initialize it for the new file.

The type of object the vnode represents is recorded by *v_type*. It is used by generic code to perform checks to ensure operations are performed on valid file system objects. Valid types are:

VNON The vnode has no type.
VREG The vnode represents a regular file.
VDIR The vnode represents a directory.
VBLK The vnode represents a block special device.
VCHR The vnode represents a character special device.
VLNK The vnode represents a symbolic link.

VSOCK
: The vnode represents a socket.

VFIFO
: The vnode represents a pipe.

VBAD
: The vnode represents a bad file (not currently used).

Vnode tag types are used by external programs only (e.g., `pstat`(8)), and should never be inspected by the kernel. Its use is deprecated since new *v_tag* values cannot be defined for loadable file systems. The *v_tag* member is read-only. Valid tag types are:

VT_NON	non file system
VT_UFS	universal file system
VT_NFS	network file system
VT_MFS	memory file system
VT_MSDOSFS	FAT file system
VT_LFS	log-structured file system
VT_LOFS	loopback file system
VT_FDESC	file descriptor file system
VT_PORTAL	portal daemon
VT_NULL	null file system layer
VT_UMAP	uid/gid remapping file system layer
VT_KERNFS	kernel interface file system
VT_PROCFS	process interface file system
VT_AFS	AFS file system
VT_ISOFS	ISO 9660 file system(s)
VT_UNION	union file system
VT_ADOSFS	Amiga file system
VT_EXT2FS	Linux's EXT2 file system
VT_CODA	Coda file system
VT_FILECORE	filecore file system
VT_NTFS	Microsoft NT's file system
VT_VFS	virtual file system
VT_OVERLAY	overlay file system
VT_SMBFS	SMB file system
VT_PTYFS	pseudo-terminal device file system
VT_TMPFS	efficient memory file system
VT_UDF	universal disk format file system
VT_SYSVBFS	systemV boot file system

All vnode locking operations use *v_vnlock*. This lock is acquired by calling `vn_lock`(9) and released by calling `VOP_UNLOCK`(9). The reason for this asymmetry is that `vn_lock`(9) is a wrapper for `VOP_LOCK`(9) with extra checks, while the unlocking step usually does not need additional checks and thus has no wrapper.

The vnode locking operation is complicated because it is used for many purposes. Sometimes it is used to bundle a series of vnode operations (see `vnodeops`(9)) into an atomic group. Many file systems rely on it to prevent race conditions in updating file system type specific data structures rather than using their own private locks. The vnode lock can operate as a multiple-reader (shared-access lock) or single-writer lock (exclusive access lock), however many current file system implementations were written assuming only single-writer locking. Multiple-reader locking functions equivalently only in the presence of big-lock SMP locking or a uni-processor machine. The lock may be held while sleeping. While the *v_vnlock* is acquired, the holder is guaranteed that the vnode will not be reclaimed or invalidated. Most file system functions require that you hold the vnode lock on entry. See `lock`(9) for details on the kernel locking API.

For leaf file systems (such as ffs, lfs, msdosfs, etc), *v_vnlock* will point to *v_lock*. For stacked file systems, *v_vnlock* will generally point to *v_vlock* of the lowest file system. Additionally, the implementation of the vnode lock is the responsibility of the individual file systems and *v_vnlock* may also be NULL indicating that a leaf node does not export a lock for vnode locking. In this case, stacked file systems (such as nullfs) must call the underlying file system directly for locking.

Each file system underlying a vnode allocates its own private area and hangs it from *v_data*.

Most functions discussed in this page that operate on vnodes cannot be called from interrupt context. The members *v_numoutput*, *v_holdcnt*, *v_dirtyblkhd*, *v_cleanblkhd*, *v_freelist*, and *v_synclist* are modified in interrupt context and must be protected by `splbio`(9) unless it is certain that there is no chance an interrupt handler will modify them. The vnode lock must not be acquired within interrupt context.

FUNCTIONS

`vcount`(*vp*)
: Calculate the total number of reference counts to a special device with vnode *vp*.

`vref`(*vp*)
: Increment *v_usecount* of the vnode *vp*. Any kernel thread system which uses a vnode (e.g., during the operation of some algorithm or to store in a data structure) should call **`vref`**().

`VREF`(*vp*)
: This function is an alias for **`vref`**().

`vrele`(*vp*)
: Decrement *v_usecount* of unlocked vnode *vp*. Any code in the system which is using a vnode should call **`vrele`**() when it is finished with the vnode. If *v_usecount* of the vnode reaches zero and *v_holdcnt* is greater than zero, the vnode is placed on the holdlist. If both *v_usecount* and *v_holdcnt* are zero, the vnode is placed on the freelist.

`vget`(*vp*, *lockflags*)
: Reclaim vnode *vp* from the freelist, increment its reference count and lock it. The argument *lockflags* specifies the `lockmgr`(9) flags used to lock the vnode. If the VXLOCK is set in *vp*'s *v_flag*, vnode *vp* is being recycled in **`vgone`**() and the calling thread sleeps until the transition is complete. When it is awakened, an error is returned to indicate that the vnode is no longer usable (possibly having been recycled to a new file system type).

`vput`(*vp*)
: Unlock vnode *vp* and decrement its *v_usecount*. Depending on the reference counts, move the vnode to the holdlist or the freelist. This operation is functionally equivalent to calling `VOP_UNLOCK`(9) followed by **`vrele`**().

`vhold`(*vp*)
: Mark the vnode *vp* as active by incrementing *vp->v_holdcnt* and moving the vnode from the freelist to the holdlist. Once on the holdlist, the vnode will not be recycled until it is released with **`holdrele`**().

`VHOLD`(*vp*)
: This function is an alias for **`vhold`**().

`holdrele`(*vp*)
: Mark the vnode *vp* as inactive by decrementing *vp->v_holdcnt* and moving the vnode from the holdlist to the freelist.

`HOLDRELE`(*vp*)
: This function is an alias for **`holdrele`**().

getnewvnode(*tag*, *mp*, *vops*, *vpp*)
Retrieve the next vnode from the freelist. **getnewvnode**() must choose whether to allocate a new vnode or recycle an existing one. The criterion for allocating a new one is that the total number of vnodes is less than the number desired or there are no vnodes on either free list. Generally only vnodes that have no buffers associated with them are recycled and the next vnode from the freelist is retrieved. If the freelist is empty, vnodes on the holdlist are considered. The new vnode is returned in the address specified by *vpp*.

The argument *mp* is the mount point for the file system requested the new vnode. Before retrieving the new vnode, the file system is checked if it is busy (such as currently unmounting). An error is returned if the file system is unmounted.

The argument *tag* is the vnode tag assigned to **vpp->v_tag*. The argument *vops* is the vnode operations vector of the file system requesting the new vnode. If a vnode is successfully retrieved zero is returned, otherwise an appropriate error code is returned.

ungetnewvnode(*vp*)
Undo the operation of **getnewvnode**(). The argument *vp* is the vnode to return to the freelist. This function is needed for VFS_VGET(9) which may need to push back a vnode in case of a locking race condition.

vrecycle(*vp*, *inter_lkp*, *l*)
Recycle the unused vnode *vp* to the front of the freelist. **vrecycle**() is a null operation if the reference count is greater than zero.

vgone(*vp*)
Eliminate all activity associated with the unlocked vnode *vp* in preparation for recycling.

vgonel(*vp*, *p*)
Eliminate all activity associated with the locked vnode *vp* in preparation for recycling.

vflush(*mp*, *skipvp*, *flags*)
Remove any vnodes in the vnode table belonging to mount point *mp*. If *skipvp* is not NULL it is exempt from being flushed. The argument *flags* is a set of flags modifying the operation of **vflush**(). If FORCECLOSE is not specified, there should not be any active vnodes and the error EBUSY is returned if any are found (this is a user error, not a system error). If FORCECLOSE is specified, active vnodes that are found are detached. If WRITECLOSE is set, only flush out regular file vnodes open for writing. SKIPSYSTEM causes any vnodes marked V_SYSTEM to be skipped.

vaccess(*type*, *file_mode*, *uid*, *gid*, *acc_mode*, *cred*)
Do access checking by comparing the file's permissions to the caller's desired access type *acc_mode* and credentials *cred*.

checkalias(*vp*, *nvp_rdev*, *mp*)
Check to see if the new vnode *vp* represents a special device for which another vnode represents the same device. If such an alias exists, the existing contents and the aliased vnode are deallocated. The caller is responsible for filling the new vnode with its new contents.

bdevvp(*dev*, *vpp*)
Create a vnode for a block device. **bdevvp**() is used for root file systems, swap areas and for memory file system special devices.

cdevvp(*dev*, *vpp*)
Create a vnode for a character device. **cdevvp**() is used for the console and kernfs special devices.

vfinddev(*dev*, *vtype*, *vpp*)
Lookup a vnode by device number. The vnode is returned in the address specified by *vpp*.

vdevgone(*int maj*, *int min*, *int minh*, *enum vtype type*)
Reclaim all vnodes that correspond to the specified minor number range *minl* to *minh* (endpoints inclusive) of the specified major *maj*.

vwakeup(*bp*)
Update outstanding I/O count *vp->v_numoutput* for the vnode *bp->b_vp* and do a wakeup if requested and *vp->vflag* has VBWAIT set.

vflushbuf(*vp*, *sync*)
Flush all dirty buffers to disk for the file with the locked vnode *vp*. The argument *sync* specifies whether the I/O should be synchronous and **vflushbuf**() will sleep until *vp->v_numoutput* is zero and *vp->v_dirtyblkhd* is empty.

vinvalbuf(*vp*, *flags*, *cred*, *l*, *slpflag*, *slptimeo*)
Flush out and invalidate all buffers associated with locked vnode *vp*. The argument *l* and *cred* specified the calling process and its credentials. The ltsleep(9) flag and timeout are specified by the arguments *slpflag* and *slptimeo* respectively. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

vtruncbuf(*vp*, *lbn*, *slpflag*, *slptimeo*)
Destroy any in-core buffers past the file truncation length for the locked vnode *vp*. The truncation length is specified by *lbn*. **vtruncbuf**() will sleep while the I/O is performed, The ltsleep(9) flag and timeout are specified by the arguments *slpflag* and *slptimeo* respectively. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

vprint(*label*, *vp*)
This function is used by the kernel to dump vnode information during a panic. It is only used if the kernel option DIAGNOSTIC is compiled into the kernel. The argument *label* is a string to prefix the information dump of vnode *vp*.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the vnode framework can be found. All pathnames are relative to /usr/src.

The vnode framework is implemented within the files sys/kern/vfs_subr.c and sys/kern/vfs_subr2.c.

SEE ALSO

intro(9), lock(9), namecache(9), namei(9), uvm(9), vattr(9), vfs(9), vfsops(9), vnodeops(9), vnsubr(9)

BUGS

The locking protocol is inconsistent. Many vnode operations are passed locked vnodes on entry but release the lock before they exit. The locking protocol is used in some places to attempt to make a series of operations atomic (e.g., access check then operation). This does not work for non-local file systems that do not support locking (e.g., NFS). The **vnode** interface would benefit from a simpler locking protocol.

NAME

vnodeops, **VOP_LOOKUP**, **VOP_CREATE**, **VOP_MKNOD**, **VOP_OPEN**, **VOP_CLOSE**, **VOP_ACCESS**, **VOP_GETATTR**, **VOP_SETATTR**, **VOP_READ**, **VOP_WRITE**, **VOP_IOCTL**, **VOP_FCNTL**, **VOP_POLL**, **VOP_KQFILTER**, **VOP_REVOKE**, **VOP_MMAP**, **VOP_FSYNC**, **VOP_SEEK**, **VOP_REMOVE**, **VOP_LINK**, **VOP_RENAME**, **VOP_MKDIR**, **VOP_RMDIR**, **VOP_SYMLINK**, **VOP_READDIR**, **VOP_READLINK**, **VOP_ABORTOP**, **VOP_INACTIVE**, **VOP_RECLAIM**, **VOP_LOCK**, **VOP_UNLOCK**, **VOP_ISLOCKED**, **VOP_BMAP**, **VOP_PRINT**, **VOP_PATHCONF**, **VOP_ADVLOCK**, **VOP_LEASE**, **VOP_WHITEOUT**, **VOP_GETPAGES**, **VOP_PUTPAGES**, **VOP_STRATEGY**, **VOP_BWRITE**, **VOP_GETEXTATTR**, **VOP_SETEXTATTR**, **VOP_LISTEXTATTR** — vnode operations

SYNOPSIS

```
#include <sys/param.h>
#include <sys/buf.h>
#include <sys/dirent.h>
#include <sys/lock.h>
#include <sys/vnode.h>
#include <sys/mount.h>
#include <sys/namei.h>
#include <sys/unistd.h>
#include <sys/fcntl.h>
#include <sys/lockf.h>
#include <sys/extattr.h>

int
VOP_LOOKUP(struct vnode *dvp, struct vnode **vpp,
    struct componentname *cnp);

int
VOP_CREATE(struct vnode *dvp, struct vnode **vpp,
    struct componentname *cnp, struct vattr *vap);

int
VOP_MKNOD(struct vnode *dvp, struct vnode **vpp, struct componentname *cnp,
    struct vattr *vap);

int
VOP_OPEN(struct vnode *vp, int mode, kauth_cred_t cred);

int
VOP_CLOSE(struct vnode *vp, int fflag, kauth_cred_t cred);

int
VOP_ACCESS(struct vnode *vp, int mode, kauth_cred_t cred);

int
VOP_GETATTR(struct vnode *vp, struct vattr *vap, kauth_cred_t cred);

int
VOP_SETATTR(struct vnode *vp, struct vattr *vap, kauth_cred_t cred);

int
VOP_READ(struct vnode *vp, struct uio *uio, int ioflag, kauth_cred_t cred);

int
VOP_WRITE(struct vnode *vp, struct uio *uio, int ioflag, kauth_cred_t cred);
```

int
VOP_IOCTL(*struct vnode *vp*, *u_long command*, *void *data*, *int fflag*, *kauth_cred_t cred*);

int
VOP_FCNTL(*struct vnode *vp*, *u_int command*, *void *data*, *int fflag*, *kauth_cred_t cred*);

int
VOP_POLL(*struct vnode *vp*, *int events*);

int
VOP_KQFILTER(*struct vnode *vp*, *struct knote *kn*);

int
VOP_REVOKE(*struct vnode *vp*, *int flags*);

int
VOP_MMAP(*struct vnode *vp*, *vm_prot_t prot*, *kauth_cred_t cred*);

int
VOP_FSYNC(*struct vnode *vp*, *kauth_cred_t cred*, *int flags*, *off_t offlo*, *off_t offhi*);

int
VOP_SEEK(*struct vnode *vp*, *off_t oldoff*, *off_t newoff*, *kauth_cred_t cred*);

int
VOP_REMOVE(*struct vnode *vp*, *struct vnode *vp*, *struct componentname *cnp*);

int
VOP_LINK(*struct vnode *dvp*, *struct vnode *vp*, *struct componentname *cnp*);

int
VOP_RENAME(*struct vnode *fdvp*, *struct vnode *fvp*, *struct componentname *fcnp*, *struct vnode *tdvp*, *struct vnode *tvp*, *struct componentname *tcnp*);

int
VOP_MKDIR(*struct vnode *dvp*, *struct vnode **vpp*, *struct componentname *cnp*, *struct vattr *vap*);

int
VOP_RMDIR(*struct vnode *dvp*, *struct vnode *vp*, *struct componentname *cnp*);

int
VOP_SYMLINK(*struct vnode *dvp*, *struct vnode **vpp*, *struct componentname *cnp*, *struct vattr *vap*, *char *target*);

int
VOP_READDIR(*struct vnode *vp*, *struct uio *uio*, *kauth_cred_t cred*, *int *eofflag*, *off_t **cookies*, *int *ncookies*);

int
VOP_READLINK(*struct vnode *vp*, *struct uio *uio*, *kauth_cred_t cred*);

int
VOP_ABORTOP(*struct vnode *dvp*, *struct componentname *cnp*);

int
VOP_INACTIVE(*struct vnode *vp*);

int
VOP_RECLAIM(*struct vnode *vp*);

int
VOP_LOCK(*struct vnode *vp*, *int flags*);

int
VOP_UNLOCK(*struct vnode *vp*, *int flags*);

int
VOP_ISLOCKED(*struct vnode *vp*);

int
VOP_BMAP(*struct vnode *vp*, *daddr_t bn*, *struct vnode **vpp*, *daddr_t *bnp*,
*int *runp*);

int
VOP_PRINT(*struct vnode *vp*);

int
VOP_PATHCONF(*struct vnode *vp*, *int name*, *register_t *retval*);

int
VOP_ADVLOCK(*struct vnode *vp*, *void *id*, *int op*, *struct flock *fl*, *int flags*);

int
VOP_LEASE(*struct vnode *vp*, *kauth_cred_t cred*, *int flag*);

int
VOP_WHITEOUT(*struct vnode *dvp*, *struct componentname *cnp*, *int flags*);

int
VOP_GETPAGES(*struct vnode *vp*, *voff_t offset*, *struct vm_page **m*,
*int *count*, *int centeridx*, *vm_prot_t access_type*, *int advice*,
int flags);

int
VOP_PUTPAGES(*struct vnode *vp*, *voff_t offlo*, *voff_t offhi*, *int flags*);

int
VOP_STRATEGY(*struct vnode *vp*, *struct buf *bp*);

int
VOP_BWRITE(*struct buf *bp*);

int
VOP_GETEXTATTR(*struct vnode *vp*, *int attrnamespace*, *const char *name*,
*struct uio *uio*, *size_t *size*, *kauth_cred_t cred*);

int
VOP_SETEXTATTR(*struct vnode *vp*, *int attrnamespace*, *const char *name*,
*struct uio *uio*, *kauth_cred_t cred*);

int
VOP_LISTEXTATTR(*struct vnode *vp*, *int attrnamespace*, *struct uio *uio*,
*size_t *size*, *kauth_cred_t cred*);

Not all header files are required for each function.

DESCRIPTION

The vnode operations vector describes what operations can be done to the file associated with the vnode. The system maintains one vnode operations vector for each file system type configured into the kernel. The vnode operations vector contains a pointer to a function for each operation supported by the file system. Many of the functions described in the vnode operations vector are closely related to their corresponding system calls. In most cases, they are called as a result of the system call associated with the operation being invoked.

Functions in the vnode operations vector are invoked using specialized macros. The following table lists the elements of the vnode operations vector, the corresponding invocation macro, and a description of the element.

Vector element	*Macro*	*Description*
int (*vop_lookup)()	VOP_LOOKUP	Lookup file name in name cache
int (*vop_create)()	VOP_CREATE	Create a new file
int (*vop_mknod)()	VOP_MKNOD	Make a new device
int (*vop_open)()	VOP_OPEN	Open a file
int (*vop_close)()	VOP_CLOSE	Close a file
int (*vop_access)()	VOP_ACCESS	Determine file accessibility
int (*vop_getattr)()	VOP_GETATTR	Get file attributes
int (*vop_setattr)()	VOP_SETATTR	Set file attributes
int (*vop_read)()	VOP_READ	Read from a file
int (*vop_write)()	VOP_WRITE	Write to a file
int (*vop_ioctl)()	VOP_IOCTL	Perform device-specific I/O
int (*vop_fcntl)()	VOP_FCNTL	Perform file control
int (*vop_poll)()	VOP_POLL	Test if poll event has occurred
int (*vop_kqfilter)()	VOP_KQFILTER	Register a knote
int (*vop_revoke)()	VOP_REVOKE	Eliminate vode activity
int (*vop_mmap)()	VOP_MMAP	Map file into user address space
int (*vop_fsync)()	VOP_FSYNC	Flush pending data to disk
int (*vop_seek)()	VOP_SEEK	Test if file is seekable
int (*vop_remove)()	VOP_REMOVE	Remove a file
int (*vop_link)()	VOP_LINK	Link a file
int (*vop_rename)()	VOP_RENAME	Rename a file
int (*vop_mkdir)()	VOP_MKDIR	Make a new directory
int (*vop_rmdir)()	VOP_RMDIR	Remove a directory
int (*vop_symlink)()	VOP_SYMLINK	Create a symbolic link
int (*vop_readdir)()	VOP_READDIR	Read directory entry
int (*vop_readlink)()	VOP_READLINK	Read contents of a symlink
int (*vop_abortop)()	VOP_ABORTOP	Abort pending operation
int (*vop_inactive)()	VOP_INACTIVE	Release the inactive vnode
int (*vop_reclaim)()	VOP_RECLAIM	Reclaim vnode for another file
int (*vop_lock)()	VOP_LOCK	Sleep until vnode lock is free
int (*vop_unlock)()	VOP_UNLOCK	Wake up process sleeping on lock
int (*vop_islocked)()	VOP_ISLOCKED	Test if vnode is locked
int (*vop_bmap)()	VOP_BMAP	Logical block number conversion
int (*vop_print)()	VOP_PRINT	Print debugging information
int (*vop_pathconf)()	VOP_PATHCONF	Return POSIX pathconf data
int (*vop_advlock)()	VOP_ADVLOCK	Advisory record locking
int (*vop_lease)()	VOP_LEASE	Validate vnode credentials
int (*vop_whiteout)()	VOP_WHITEOUT	Whiteout vnode

int (*vop_getpages)()	VOP_GETPAGES	Read VM pages from file
int (*vop_putpages)()	VOP_PUTPAGES	Write VM pages to file
int (*vop_strategy)()	VOP_STRATEGY	Read/write a file system buffer
int (*vop_bwrite)()	VOP_BWRITE	Write a file system buffer
int (*vop_getextattr)()	VOP_GETEXTATTR	Get extended attribute
int (*vop_setextattr)()	VOP_SETEXTATTR	Set extended attribute
int (*vop_listextattr)()	VOP_LISTEXTATTR	List extended attributes

The implementation details of the vnode operations vector are not quite what is described here.

If the file system type does not support a specific operation, it must nevertheless assign an appropriate function in the vnode operations vector to do the minimum required of it. In most cases, such functions either do nothing or return an error value to the effect that it is not supported.

Many of the functions in the vnode operations vector take a componentname structure. It is used to encapsulate many parameters into a single function argument. It has the following structure:

```
struct componentname {
        /*
         * Arguments to lookup.
         */
        uint32_t cn_nameiop;    /* namei operation */
        uint32_t cn_flags;      /* flags to namei */
        kauth_cred_t cn_cred;   /* credentials */
        /*
         * Shared between lookup and commit routines.
         */
        char    *cn_pnbuf;      /* pathname buffer */
        const char *cn_nameptr; /* pointer to looked up name */
        size_t  cn_namelen;     /* length of looked up component */
        u_long  cn_hash;        /* hash value of looked up name */
        size_t  cn_consume;     /* chars to consume in lookup() */
};
```

The top half of the structure is used exclusively for the pathname lookups using **VOP_LOOKUP**() and is initialized by the caller. The semantics of the lookup are affected by the lookup operation specified in *cn_nameiop* and the flags specified in *cn_flags*. Valid operations are:

LOOKUP
: perform name lookup only

CREATE
: set up for file creation

DELETE
: set up for file deletion

RENAME
: set up for file renaming

OPMASK
: mask for operation

Valid values for *cn->cn_flags* are:

LOCKLEAF lock inode on return

LOCKPARENT
: want parent vnode returned locked

NOCACHE	name must not be left in name cache (see namecache(9))
FOLLOW	follow symbolic links
NOFOLLOW	do not follow symbolic links (pseudo)
MODMASK	mask of operational modifiers

No vnode operations may be called from interrupt context. Most operations also require the vnode to be locked on entry. To prevent deadlocks, when acquiring locks on multiple vnodes, the lock of parent directory must be acquired before the lock on the child directory.

Vnode operations for a file system type generally should not be called directly from the kernel, but accessed indirectly through the high-level convenience functions discussed in vnsubr(9).

FUNCTIONS

VOP_LOOKUP(*dvp*, *vpp*, *cnp*)

Lookup a single pathname component in a given directory. The argument *dvp* is the locked vnode of the directory to search and *cnp* is the pathname component to be searched for. If the pathname component is found, the address of the resulting locked vnode is returned in *vpp*. The operation specified in *cnp->cn_nameiop* gives **VOP_LOOKUP**() hints about the reason for requesting the lookup and uses it to cache file system type specific information in the vnode for subsequent operations.

There are three types of lookups: ".", ".." (ISDOTDOT), and other. If the pathname component being searched for is ".", then *dvp* has an extra reference added to it and it is returned in **vpp*. If the pathname component being search for is ".." (ISDOTDOT), *dvp* is unlocked, the ".." node is locked and then *dvp* is relocked. This process preserves the protocol of always locking nodes from root downward and prevents deadlock. For other pathname components, **VOP_LOOKUP**() checks the accessibility of the directory and searches the name cache for the pathname component. See namecache(9). If the pathname is not found in the name cache, the directory is searched for the pathname. The resulting locked vnode is returned in *vpp*. *dvp* is always returned locked.

On failure **vpp* is NULL, and **dvp* is left locked. If the operation is successful **vpp* is locked and zero is returned. Typically, if **vpp* and *dvp* are the same vnode the caller will need to release twice (decrement the reference count) and unlock once.

VOP_CREATE(*dvp*, *vpp*, *cnp*, *vap*)

Create a new file in a given directory. The argument *dvp* is the locked vnode of the directory to create the new file in and *cnp* is the pathname component of the new file. The argument *vap* specifies the attributes that the new file should be created with. If the file is successfully created, the address of the resulting locked vnode is returned in *vpp* and zero is returned. Regardless of the return value, the directory vnode *dvp* will be unlocked on return.

This function is called after **VOP_LOOKUP**() when a file is being created. Normally, **VOP_LOOKUP**() will have set the SAVENAME flag in *cnp->cn_flags* to keep the memory pointed to by *cnp->cn_pnbuf* valid. If an error is detected when creating the file, this memory is released. If the file is created successfully it will be released unless the SAVESTART flags in specified in *cnp->cn_flags*.

VOP_MKNOD(*dvp*, *vpp*, *cnp*, *vap*)

Make a new device-special file in a given directory. The argument *dvp* is the locked vnode of the directory to create the new device-special file in and *cnp* is the pathname component of the new device-special file. The argument *vap* specifies the attributes that the new device-special file should be created with. If the file is successfully created, the address of the resulting locked vnode is returned in *vpp* and zero is returned.

This function is called after **VOP_LOOKUP**() when a device-special file is being created. Normally, **VOP_LOOKUP**() will have set the SAVENAME flag in *cnp->cn_flags* to keep the memory pointed to by *cnp->cn_pnbuf* valid. If an error is detected when creating the device-special file, this memory is released. If the device-special file is created successfully it will be released unless the SAVESTART flags in specified in *cnp->cn_flags*.

VOP_OPEN(*vp*, *mode*, *cred*)
Open a file. The argument *vp* is the vnode of the file to open and *mode* specifies the access mode required by the calling process. The calling credentials are specified by *cred*. The access mode is a set of flags, including FREAD, FWRITE, O_NONBLOCK, O_APPEND, etc. **VOP_OPEN**() must be called before a file can be accessed by a thread. The vnode reference count is incremented.

VOP_OPEN() expects the vnode *vp* to be locked on entry and will leave it locked on return. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

VOP_CLOSE(*vp*, *fflag*, *cred*)
Close a file. The argument *vp* is the vnode of the file to close and *fflag* specifies the access mode by the calling process. The possible flags are FREAD, FWRITE and FNONBLOCK. The calling credentials are specified by *cred*. **VOP_CLOSE**() frees resources allocated by **VOP_OPEN**().

The vnode *vp* will be locked on entry and should remain locked on return.

VOP_ACCESS(*vp*, *mode*, *cred*)
Determine the accessibility (permissions) of the file against the specified credentials. The argument *vp* is the vnode of the file to check, *mode* is the type of access required and *cred* contains the user credentials to check. The argument *mode* is a mask which can contain VREAD, VWRITE or VEXEC. If the file is accessible in the specified way, zero is returned, otherwise an appropriate error code is returned.

The vnode *vp* will be locked on entry and should remain locked on return.

VOP_GETATTR(*vp*, *vap*, *cred*)
Get specific vnode attributes on a file. The argument *vp* is the vnode of the file to get the attributes for. The argument *cred* specifies the calling credentials. **VOP_GETATTR**() uses the file system type specific data object *vp->v_data* to reference the underlying file attributes. Attributes associated with the file are collected by setting the required attribute bits in *vap->va_mask*. The attributes are returned in *vap*. Attributes which are not available are set to the value VNOVAL.

For more information on vnode attributes see vattr(9).

VOP_SETATTR(*vp*, *vap*, *cred*)
Set specific vnode attributes on a file. The argument *vp* is the locked vnode of the file to set the attributes for. The argument *cred* specifies the calling credentials. **VOP_SETATTR**() uses the file system type specific data object *vp->v_data* to reference the underlying file attributes. The new attributes are defined in *vap*. Attributes associated with the file are set by setting the required attribute bits in *vap->va_mask*. Attributes which are not being modified by **VOP_SETATTR**() should be set to the value VNOVAL. If the operation is successful zero is returned, otherwise an appropriate error is returned.

For more information on vnode attributes see vattr(9).

VOP_READ(*vp*, *uio*, *ioflag*, *cred*)
Read the contents of a file. The argument *vp* is the vnode of the file to read from, *uio* is the location to read the data into, *ioflag* is a set of flags and *cred* are the credentials of the call-

ing process.

The *ioflag* argument is used to give directives and hints to the file system. When attempting a read, the high 16 bits are used to provide a read-ahead hint (in unit of file system blocks) that the file system should attempt. The low 16 bits are a bit mask which can contain the following flags:

IO_UNIT	do I/O as atomic unit
IO_APPEND	append write to end
IO_SYNC	sync I/O file integrity completion
IO_NODELOCKED	underlying node already locked
IO_NDELAY	FNDELAY flag set in file table
IO_DSYNC	sync I/O data integrity completion
IO_ALTSEMANTICS	use alternate I/O semantics
IO_NORMAL	operate on regular data
IO_EXT	operate on extended attributes
IO_DIRECT	do not buffer data in the kernel

Zero is returned on success, otherwise an error is returned. The vnode should be locked on entry and remains locked on exit.

VOP_WRITE(*vp*, *uio*, *ioflag*, *cred*)

Write to a file. The argument *vp* is the vnode of the file to write to, *uio* is the location of the data to write, *ioflag* is a set of flags and *cred* are the credentials of the calling process.

The *ioflag* argument is used to give directives and hints to the file system. The low 16 bits are a bit mask which can contain the same flags as **VOP_READ**().

Zero is returned on success, otherwise an error is returned. The vnode should be locked on entry and remains locked on exit.

VOP_IOCTL(*vp*, *command*, *data*, *fflag*, *cred*)

Perform device-specific I/O. The argument *vp* is the locked vnode of the file, normally representing a device. The argument *command* specifies the device-specific operation to perform and *cnp* provides extra data for the specified operation. The argument *fflags* is a set of flags. The argument *cred* is the caller's credentials. If the operation is successful, zero is returned, otherwise an appropriate error code is returned.

Most file systems do not supply a function for **VOP_IOCTL**(). This function implements the ioctl(2) system call.

VOP_FCNTL(*vp*, *command*, *data*, *fflag*, *cred*)

Perform file control. The argument *vp* is the locked vnode of the file. The argument *command* specifies the operation to perform and *cnp* provides extra data for the specified operation. The argument *fflags* is a set of flags. The argument *cred* is the caller's credentials. If the operation is successful, zero is returned, otherwise an appropriate error code is returned.

VOP_POLL(*vp*, *events*)

Test if a poll event has occurred. The argument *vp* is the vnode of the file to poll. It returns any events of interest as specified by *events* that may have occurred for the file. The argument *events* is a set of flags as specified by poll(2).

VOP_KQFILTER(*vp*, *kn*)

Register a knote *kn* with the vnode *vn*. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

VOP_REVOKE(*vp*, *flags*)
Eliminate all activity associated with the vnode *vp*. The argument *flags* is a set of flags. If REVOKEALL is set in *flags* all vnodes aliased to the vnode *vp* are also eliminated. If the operation is successful zero is returned, otherwise an appropriate error is returned.

VOP_MMAP(*vp*, *prot*, *cred*)
Inform file system that *vp* is in the process of being memory mapped. The argument *prot* specifies the vm access protection the vnode is going to be mapped with. The argument *cred* is the caller's credentials. If the file system allows the memory mapping, zero is returned, otherwise an appropriate error code is returned.

Most file systems do not supply a function for **VOP_MMAP**() and use **genfs_mmap**() to default for success. Only file systems which do not integrate with the page cache at all typically want to disallow memory mapping.

VOP_FSYNC(*vp*, *cred*, *flags*, *offlo*, *offhi*)
Flush pending data buffers for a file to disk. The argument *vp* is the locked vnode of the file for flush. The argument *cred* is the caller's credentials. The argument *flags* is a set of flags. If FSYNC_WAIT is specified in *flags*, the function should wait for I/O to complete before returning. The argument *offlo* and *offhi* specify the range of file to flush. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

This function implements the sync(2) and fsync(2) system calls.

VOP_SEEK(*vp*, *oldoff*, *newoff*, *cred*)
Test if the file is seekable for the specified offset *newoff*. The argument *vp* is the locked vnode of the file to test. For most file systems this function simply tests if *newoff* is valid. If the specified *newoff* is less than zero, the function returns error code EINVAL.

VOP_REMOVE(*dvp*, *vp*, *cnp*)
Remove a file. The argument *dvp* is the locked vnode of the directory to remove the file from and *vp* is the locked vnode of the file to remove. The argument *cnp* is the pathname component about the file to remove. If the operation is successful zero is returned, otherwise an appropriate error code is returned. Both *dvp* and *vp* are locked on entry and are to be unlocked before returning.

VOP_LINK(*dvp*, *vp*, *cnp*)
Link to a file. The argument *dvp* is the locked node of the directory to create the new link and *vp* is the vnode of the file to be linked. The argument *cnp* is the pathname component of the new link. If the operation is successful zero is returned, otherwise an error code is returned. The directory vnode *dvp* should be locked on entry and will be released and unlocked on return. The vnode *vp* should not be locked on entry and will remain unlocked on return.

VOP_RENAME(*fdvp*, *fvp*, *fcnp*, *tdvp*, *tvp*, *tcnp*)
Rename a file. The argument *fdvp* is the vnode of the old parent directory containing in the file to be renamed and *fvp* is the vnode of the file to be renamed. The argument *fcnp* is the pathname component about the file to be renamed. The argument *tdvp* is the vnode of the new directory of the target file and *tvp* is the vnode of the target file (if it exists). The argument *tcnp* is the pathname component about the file's new name. If the operation is successful zero is returned, otherwise and error code is returned.

The source directory and file vnodes should be unlocked and their reference counts should be incremented before entry. The target directory and file vnodes should both be locked on entry. **VOP_RENAME**() updates the reference counts prior to returning.

VOP_MKDIR(*dvp*, *vpp*, *cnp*, *vap*)

Make a new directory in a given directory. The argument *dvp* is the locked vnode of the directory to create the new directory in and *cnp* is the pathname component of the new directory. The argument *vap* specifies the attributes that the new directory should be created with. If the file is successfully created, the address of the resulting locked vnode is returned in *vpp* and zero is returned.

This function is called after **VOP_LOOKUP**() when a directory is being created. Normally, **VOP_LOOKUP**() will have set the SAVENAME flag in *cnp->cn_flags* to keep the memory pointed to by *cnp->cn_pnbuf* valid. If an error is detected when creating the directory, this memory is released. If the directory is created successfully it will be released unless the SAVESTART flags in specified in *cnp->cn_flags*.

VOP_RMDIR(*dvp*, *vp*, *cnp*)

Remove a directory in a given directory. The argument *dvp* is the locked vnode of the directory to remove the directory from and *vp* is the locked vnode of the directory to remove. The argument *cnp* is the pathname component of the directory. Zero is returned on success, otherwise an error code is returned. Both *dvp* and *vp* should be locked on entry and will be released and unlocked on return.

VOP_SYMLINK(*dvp*, *vpp*, *cnp*, *vap*, *target*)

Create a symbolic link in a given directory. The argument *dvp* is the locked vnode of the directory to create the symbolic link in and *cnp* is the pathname component of the symbolic link. The argument *vap* specifies the attributes that the symbolic link should be created with and *target* specifies the pathname of the target of the symbolic link. If the symbolic link is successfully created, the address of the resulting locked vnode is returned in *vpp* and zero is returned.

This function is called after **VOP_LOOKUP**() when a symbolic link is being created. Normally, **VOP_LOOKUP**() will have set the SAVENAME flag in *cnp->cn_flags* to keep the memory pointed to by *cnp->cn_pnbuf* valid. If an error is detected when creating the symbolic link, this memory is released. If the symbolic link is created successfully it will be released unless the SAVESTART flags in specified in *cnp->cn_flags*.

VOP_READDIR(*vp*, *uio*, *cred*, *eofflag*, *cookies*, *ncookies*)

Read directory entry. The argument *vp* is the vnode of the directory to read the contents of and *uio* is the destination location to read the contents into. The argument *cred* is the caller's credentials. The argument *eofflag* is the pointer to a flag which is set by **VOP_READDIR**() to indicate an end-of-file condition. If *eofflag* is NULL, the end-of-file condition is not returned. The arguments *cookies* and *ncookies* specify the addresses for the list and number of directory seek cookies generated for NFS. Both *cookies* and *ncookies* should be NULL if they aren't required to be returned by **VOP_READDIR**(). The directory contents are read into struct dirent structures and *uio->uio_offset* is set to the offset of the next unread directory entry. This offset may be used in a following invocation to continue a sequential read of the directory contents. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

The directory should be locked on entry and will remain locked on return.

In case *ncookies* and *cookies* are supplied, one cookie should be returned per directory entry. The value of the cookie for each directory entry should be the offset within the directory where the on-disk version of the following directory entry starts. That is, for each directory entry *i*, the corresponding cookie should refer to the offset of directory entry *i + 1*.

Note that the *cookies* array must be allocated by the callee using the M_TEMP malloc type as callers of **VOP_READDIR**() must be able to free the allocation.

VOP_READLINK(*vp*, *uio*, *cred*)
Read the contents of a symbolic link. The argument *vp* is the locked vnode of the symlink and *uio* is the destination location to read the contents into. The argument *cred* is the credentials of the caller. If the operation is successful zero is returned, otherwise an error code is returned.

The vnode should be locked on entry and will remain locked on return.

VOP_ABORTOP(*dvp*, *cnp*)
Abort pending operation on vnode *dvp* and free resources allocated in *cnp*.

This operation is rarely implemented in file systems and **genfs_abortop**() is typically used instead.

VOP_INACTIVE(*vp*)
Release the inactive vnode. **VOP_INACTIVE**() is called when the kernel is no longer using the vnode. This may be because the reference count reaches zero or it may be that the file system is being forcibly unmounted while there are open files. It can be used to reclaim space for open but deleted files. The argument *vp* is the locked vnode to be released. If the operation is successful zero is returned, otherwise an appropriate error code is returned. The vnode *vp* must be locked on entry, and will be unlocked on return.

VOP_RECLAIM(*vp*)
Reclaim the vnode for another file system. **VOP_RECLAIM**() is called when a vnode is being reused for a different file system. Any file system specific resources associated with the vnode should be freed. The argument *vp* is the vnode to be reclaimed. If the operation is successful zero is returned, otherwise an appropriate error code is returned. The vnode *vp* should not be locked on entry, and will remain unlocked on return.

VOP_LOCK(*vp*, *flags*)
Sleep until vnode lock is free. The argument *vp* is the vnode of the file to be locked. The argument *flags* is a set of lockmgr(9) flags. If the operation is successful zero is returned, otherwise an appropriate error code is returned. **VOP_LOCK**() is used to serialize access to the file system such as to prevent two writes to the same file from happening at the same time. Kernel code should use vn_lock(9) to lock a vnode rather than calling **VOP_LOCK**() directly.

VOP_UNLOCK(*vp*, *flags*)
Wake up process sleeping on lock. The argument *vp* is the vnode of the file to be unlocked. The argument *flags* is a set of lockmgr(9) flags. If the operation is successful zero is returned, otherwise an appropriate error code is returned. **VOP_UNLOCK**() is used to serialize access to the file system such as to prevent two writes to the same file from happening at the same time.

VOP_ISLOCKED(*vp*)
Test if the vnode *vp* is locked. A non-zero value is returned if the vnode is not locked, otherwise zero is returned.

VOP_BMAP(*vp*, *bn*, *vpp*, *bnp*, *runp*)
Convert the logical block number *bn* of a file specified by vnode *vp* to its physical block number on the disk. The physical block is returned in *bnp*. In case the logical block is not allocated, −1 is used.

If *vpp* is not NULL, the vnode of the device vnode for the file system is returned in the address specified by *vpp*. If *runp* is not NULL, the number of contiguous blocks starting from the next block after the queried block will be returned in *runp*.

VOP_PRINT(*vp*)
Print debugging information. The argument *vp* is the vnode to print. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

VOP_PATHCONF(*vp*, *name*, *retval*)

Implement POSIX pathconf(2) and fpathconf(2) support. The argument *vp* is the locked vnode to get information about. The argument *name* specified the type of information to return. The information is returned in the address specified by *retval*. Valid values for *name* are:

_PC_LINK_MAX	return the maximum number of links to a file
_PC_NAME_MAX	return the maximum number of bytes in a file name
_PC_PATH_MAX	return the maximum number of bytes in a pathname
_PC_PIPE_BUF	return the maximum number of bytes which will be written atomically to a pipe
_PC_CHOWN_RESTRICTED	return 1 if appropriate privileges are required for the chown(2) system call, otherwise zero
_PC_NO_TRUNC	return if file names longer than KERN_NAME_MAX are truncated

If *name* is recognized, **retval* is set to the specified value and zero is returned, otherwise an appropriate error is returned.

VOP_ADVLOCK(*vp*, *id*, *op*, *fl*, *flags*)

Manipulate Advisory record locks on a vnode. The argument *vp* is the vnode on which locks are manipulated. The argument *id* is the id token which is changing the lock and *op* is the fcntl(2) operation to perform. Valid values are:

F_SETLK set lock

F_GETLK
: get the first conflicted lock

F_UNLCK
: clear lock

The argument *fl* is a description of the lock. In the case of SEEK_CUR, The caller should add the current file offset to fl->l_start beforehand. **VOP_ADVLOCK**() treats SEEK_CUR as SEEK_SET.

The argument *flags* is the set of flags. Valid values are:

F_WAIT wait until lock is granted

F_FLOCK
: use flock(2) semantics for lock

F_POSIX use POSIX semantics for lock

If the operation is successful zero is returned, otherwise an appropriate error is returned.

VOP_LEASE(*vp*, *cred*, *flags*)

Validate vnode credentials and operation type. The argument *vp* is the locked vnode of the file to validate credentials *cred*. The argument *flags* specifies the operation flags. If the operation is successful zero is returned, otherwise an appropriate error code is returned. The vnode must be locked on entry and remains locked on return.

VOP_WHITEOUT(*dvp*, *cnp*, *flags*)

Whiteout pathname component in directory with vnode *dvp*. The argument *cnp* specifies the pathname component to whiteout.

VOP_GETPAGES(*vp*, *offset*, *m*, *count*, *centeridx*, *access_type*, *advice*, *flags*)

Read VM pages from file. The argument *vp* is the locked vnode to read the VM pages from. The argument *offset* is offset in the file to start accessing and *m* is an array of VM pages. The argument *count* points a variable that specifies the number of pages to read. If the operation is

successful zero is returned, otherwise an appropriate error code is returned. If PGO_LOCKED is specified in *flags*, **VOP_GETPAGES**() might return less pages than requested. In that case, the variable pointed to by *count* will be updated.

This function is primarily used by the page-fault handing mechanism.

VOP_PUTPAGES(*vp*, *offlo*, *offhi*, *flags*)
Write modified (dirty) VM pages to file. The argument *vp* is the vnode to write the VM pages to. The vnode's vm object lock (*v_uobj.vmobjlock*) must be held by the caller and will be released upon return. The arguments *offlo* and *offhi* specify the range of VM pages to write. In case *offhi* is given as 0, all pages at and after the start offset *offlo* belonging the vnode *vp* will be written. The argument *flags* controls the behavior of the routine and takes the vm pager's flags (PGO_ -prefixed). If the operation is successful zero is returned, otherwise an appropriate error code is returned.

The function is primarily used by the pageout handling mechanism and is commonly implemented indirectly by **genfs_putpages**() with the help of **VOP_STRATEGY**() and **VOP_BMAP**().

VOP_STRATEGY(*vp*, *bp*)
Read/write a file system buffer. The argument *vp* is the vnode to read/write to. The argument *bp* is the buffer to be read or written. **VOP_STRATEGY**() will either read or write data to the file depending on the value of *bp->b_flags*. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

VOP_BWRITE(*bp*)
Write a file system buffer. The argument *bp* specifies the buffer to be written. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

VOP_GETEXTATTR(*vp*, *attrnamespace*, *name*, *uio*, *size*, *cred*)
Get an extended attribute. The argument *vp* is the locked vnode of the file or directory from which to retrieve the attribute. The argument *attrnamespace* specifies the extended attribute namespace. The argument *name* is a nul-terminated character string naming the attribute to retrieve. The argument *uio*, if not NULL, specifies where the extended attribute value is to be written. The argument *size*, if not NULL, will contain the number of bytes required to read all of the attribute data upon return. In most cases, *uio* will be NULL when *size* is not, and vice versa. The argument *cred* specifies the user credentials to use when authorizing the request.

VOP_SETEXTATTR(*vp*, *attrnamespace*, *name*, *uio*, *cred*)
Set an extended attribute. The argument *vp* is the locked vnode of the file or directory to which to store the attribute. The argument *namespace* specifies the extended attribute namespace. The argument *name* is a nul-terminated character string naming the attribute to store. The argument *uio* specifies the source of the extended attribute data. The argument *cred* specifies the user credentials to use when authorizing the request.

VOP_LISTEXTATTR(*vp*, *attrnamespace*, *uio*, *size*, *cred*)
Retrieve the list of extended attributes. The argument *vp* is the locked vnode of the file or directory whose attributes are to be listed. The argument *attrnamespace* specifies the extended attribute namespace. The argument *uio*, if not NULL, specifies where the extended attribute list is to be written. The argument *size*, if not NULL, will contain the number of bytes required to read all of the attribute names upon return. In most cases, *uio* will be NULL when *size* is not, and vice versa. The argument *cred* specifies the user credentials to use when authorizing the request.

ERRORS

[ENOATTR]	The requested attribute is not defined for this vnode.
[ENOTDIR]	The vnode does not represent a directory.
[ENOENT]	The component was not found in the directory.
[ENOSPC]	The file system is full.
[EDQUOT]	Quota exceeded.
[EACCES]	Access for the specified operation is denied.
[EJUSTRETURN]	A CREATE or RENAME operation would be successful.
[EPERM]	an attempt was made to change an immutable file
[ENOTEMPTY]	attempt to remove a directory which is not empty
[EINVAL]	attempt to read from an illegal offset in the directory; unrecognized input
[EIO]	a read error occurred while reading the directory or reading the contents of a symbolic link
[EROFS]	the file system is read-only

SEE ALSO

extattr(9), intro(9), lock(9), namei(9), vattr(9), vfs(9), vfsops(9), vnode(9)

HISTORY

The vnode operations vector, its functions and the corresponding macros appeared in 4.3BSD.

NAME

vnsubr, **vn_bwrite**, **vn_close**, **vn_default_error**, **vn_isunder**, **vn_lock**, **vn_markexec**, **vn_marktext**, **vn_rdwr**, **vn_restorerecurse**, **vn_setrecurse**, **vn_open**, **vn_stat**, **vn_writechk** — high-level convenience functions for vnode operations

SYNOPSIS

```
#include <sys/param.h>
#include <sys/lock.h>
#include <sys/vnode.h>

int
vn_bwrite(void *ap);

int
vn_close(struct vnode *vp, int flags, kauth_cred_t cred);

int
vn_default_error(void *v);

int
vn_isunder(struct vnode *dvp, struct vnode *rvp, struct lwp *l);

int
vn_lock(struct vnode *vp, int flags);

void
vn_markexec(struct vnode *vp);

void
vn_marktext(struct vnode *vp);

u_int
vn_setrecurse(struct vnode *vp);

void
vn_restorerecurse(struct vnode *vp, u_int flags);

int
vn_open(struct nameidata *ndp, int fmode, int cmode);

int
vn_rdwr(enum uio_rw rw, struct vnode *vp, void *base, int len, off_t offset,
    enum uio_seg segflg, int ioflg, kauth_cred_t cred, size_t *aresid,
    struct lwp *l);

int
vn_readdir(file_t *fp, char *buf, int segflg, u_int count, int *done,
    struct lwp *l, off_t **cookies, int *ncookies);

int
vn_stat(struct vnode *vp, struct stat *sb, struct lwp *l);

int
vn_writechk(struct vnode *vp);
```

DESCRIPTION

The high-level functions described in this page are convenience functions for simplified access to the vnode operations described in vnodeops(9).

FUNCTIONS

vn_bwrite(*ap*)
Common code for block write operations.

vn_close(*vp*, *flags*, *cred*)
Common code for a vnode close. The argument *vp* is the unlocked vnode of the vnode to close. **vn_close**() simply locks the vnode, invokes the vnode operation VOP_CLOSE(9) and calls **vput**() to return the vnode to the freelist or holdlist. Note that **vn_close**() expects an unlocked, referenced vnode and will dereference the vnode prior to returning. If the operation is successful zero is returned, otherwise an appropriate error is returned.

vn_default_error(*v*)
A generic "default" routine that just returns error. It is used by a file system to specify unsupported operations in the vnode operations vector.

vn_isunder(*dvp*, *rvp*, *l*)
Common code to check if one directory specified by the vnode *rvp* can be found inside the directory specified by the vnode *dvp*. The argument *l* is the calling process. **vn_isunder**() is intended to be used in chroot(2), chdir(2), fchdir(2), etc., to ensure that chroot(2) actually means something. If the operation is successful zero is returned, otherwise 1 is returned.

vn_lock(*vp*, *flags*)
Common code to acquire the lock for vnode *vp*. The argument *flags* specifies the lockmgr(9) flags used to lock the vnode. If the operation is successful zero is returned, otherwise an appropriate error code is returned. The vnode interlock *v_interlock* is released on return.

vn_lock() must not be called when the vnode's reference count is zero. Instead, vget(9) should be used.

vn_markexec(*vp*)
Common code to mark the vnode *vp* as containing executable code of a running process.

vn_marktext(*vp*)
Common code to mark the vnode *vp* as being the text of a running process.

vn_setrecurse(*vp*)
Common code to enable LK_CANRECURSE on the vnode lock for vnode *vp*. **vn_setrecurse**() returns the new lockmgr(9) flags after the update.

vn_restorerecurse(*vp*, *flags*)
Common code to restore the vnode lock flags for the vnode *vp*. It is called when done with **vn_setrecurse**().

vn_open(*ndp*, *fmode*, *cmode*)
Common code for vnode open operations. The pathname is described in the nameidata pointer (see namei(9)). The arguments *fmode* and *cmode* specify the open(2) file mode and the access permissions for creation. **vn_open**() checks permissions and invokes the VOP_OPEN(9) or VOP_CREATE(9) vnode operations. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

vn_rdwr(*rw*, *vp*, *base*, *len*, *offset*, *segflg*, *ioflg*, *cred*, *aresid*, *l*)
Common code to package up an I/O request on a vnode into a uio and then perform the I/O. The argument *rw* specifies whether the I/O is a read (UIO_READ) or write (UIO_WRITE) operation. The unlocked vnode is specified by *vp*. The arguments *l* and *cred* are the calling lwp and its credentials. The remaining arguments specify the uio parameters. For further information on these parameters see uiomove(9).

vn_readdir(*fp*, *buf*, *segflg*, *count*, *done*, *l*, *cookies*, *ncookies*)
Common code for reading the contents of a directory. The argument *fp* is the file structure, *buf* is the buffer for placing the struct dirent structures. The arguments *cookies* and *ncookies* specify the addresses for the list and number of directory seek cookies generated for NFS. Both *cookies* and *ncookies* should be NULL is they aren't required to be returned by **vn_readdir**(). If the operation is successful zero is returned, otherwise an appropriate error code is returned.

vn_stat(*vp*, *sb*, *l*)
Common code for a vnode stat operation. The vnode is specified by the argument *vp*, and *sb* is the buffer to return the stat information. The argument *l* is the calling lwp. **vn_stat**() basically calls the vnode operation VOP_GETATTR(9) and transfers the contents of a vattr structure into a struct stat. If the operation is successful zero is returned, otherwise an appropriate error code is returned.

vn_writechk(*vp*)
Common code to check for write permission on the vnode *vp*. A vnode is read-only if it is in use as a process's text image. If the vnode is read-only ETEXTBSY is returned, otherwise zero is returned to indicate that the vnode can be written to.

ERRORS

[EBUSY] The LK_NOWAIT flag was set and **vn_lock**() would have slept.

[ENOENT] The vnode has been reclaimed and is dead. This error is only returned if the LK_RETRY flag is not passed to **vn_lock**().

[ETXTBSY] Cannot write to a vnode since is a process's text image.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the vnode framework can be found. All pathnames are relative to /usr/src.

The high-level convenience functions are implemented within the files sys/kern/vfs_vnops.c and sys/sys/vnode.h.

SEE ALSO

file(9), intro(9), lock(9), namei(9), vattr(9), vfs(9), vnode(9), vnodeops(9)

NAME

wdc — machine-independent IDE/ATAPI driver

SYNOPSIS

```
#include <dev/ata/atavar.h>
#include <sys/dev/ic/wdcvar.h>

int
wdcprobe(struct channel_softc * chp);

void
wdcattach(struct channel_softc * chp);
```

DESCRIPTION

The **wdc** driver provides the machine independent core functions for driving IDE devices. IDE devices-specific drivers (wd(4) or atapibus(4)) will use services provided by **wdc**.

The machine-dependent bus front-end provides informations to **wdc** with the *wdc_softc* and *channel_softc* structures. The first one defines global controller properties, and the second contains per-channel informations. **wdc** returns informations about the attached devices in the *ata_drive_datas* structure.

```
struct wdc_softc { /* Per controller state */
        struct device sc_dev;
        int           cap;
#define WDC_CAPABILITY_DATA16 0x0001
#define WDC_CAPABILITY_DATA32 0x0002
#define WDC_CAPABILITY_MODE   0x0004
#define WDC_CAPABILITY_DMA    0x0008
#define WDC_CAPABILITY_UDMA   0x0010
#define WDC_CAPABILITY_HWLOCK 0x0020
#define WDC_CAPABILITY_ATA_NOSTREAM 0x0040
#define WDC_CAPABILITY_ATAPI_NOSTREAM 0x0080
#define WDC_CAPABILITY_NO_EXTRA_RESETS 0x0100
#define WDC_CAPABILITY_PREATA 0x0200
#define WDC_CAPABILITY_IRQACK 0x0400
#define WDC_CAPABILITY_SINGLE_DRIVE 0x0800
#define WDC_CAPABILITY_NOIRQ  0x1000
#define WDC_CAPABILITY_SELECT  0x2000
        uint8_t      pio_mode;
        uint8_t      dma_mode;
        int nchannels;
        struct channel_softc *channels;

        void           *dma_arg;
        int           (*dma_init)(void *, int, int, void *, size_t, int);
        void          (*dma_start)(void *, int, int, int);
        int           (*dma_finish)(void *, int, int, int);
#define WDC_DMA_READ 0x01
#define WDC_DMA_POLL 0x02

        int           (*claim_hw)(void *, int);
        void           (*free_hw)(void *);
};
```

```
struct channel_softc { /* Per channel data */
        int channel;
        struct wdc_softc *wdc;
        bus_space_tag_t       cmd_iot;
        bus_space_handle_t    cmd_ioh;
        bus_space_tag_t       ctl_iot;
        bus_space_handle_t    ctl_ioh;
        bus_space_tag_t       data32iot;
        bus_space_handle_t    data32ioh;
        int ch_flags;
#define WDCF_ACTIVE   0x01
#define WDCF_IRQ_WAIT 0x10
        uint8_t ch_status;
        uint8_t ch_error;
        struct ata_drive_datas ch_drive[2];
        struct channel_queue *ch_queue;
};

struct ata_drive_datas {
    uint8_t drive;
    uint8_t drive_flags;
#define DRIVE_ATA   0x01
#define DRIVE_ATAPI 0x02
#define DRIVE (DRIVE_ATA|DRIVE_ATAPI)
#define DRIVE_CAP32 0x04
#define DRIVE_DMA   0x08
#define DRIVE_UDMA  0x10
#define DRIVE_MODE 0x20
    uint8_t PIO_mode;
    uint8_t DMA_mode;
    uint8_t UDMA_mode;
    uint8_t state;

    struct device *drv_softc;
    void* chnl_softc;
};
```

The bus front-end needs to fill in the following elements of *wdc_softc*:

cap	supports one or more of the WDC_CAPABILITY flags
nchannels	number of channels supported by this controller
channels	array of *struct channel_softc* of size *nchannels* properly initialised

The following elements are optional:

pio_mode
dma_mode
dma_arg
dma_init
dma_start
dma_finish
claim_hw
free_hw

The *WDC_CAPABILITY_DATA16* and *WDC_CAPABILITY_DATA32* flags informs **wdc** whether the controller supports 16- or 32-bit I/O accesses on the data port. If both are set, a test will be done for each drive using the ATA or ATAPI IDENTIFY command, to automatically select the working mode.

The *WDC_CAPABILITY_DMA* and *WDC_CAPABILITY_UDMA* flags are set for controllers supporting the DMA and Ultra-DMA modes. The bus front-end needs to provide the **dma_init**(), **dma_start**() and **dma_finish**() functions. **dma_init**() is called just before issuing a DMA command to the IDE device. The arguments are, respectively: *dma_arg*, the channel number, the drive number on this channel, the virtual address of the DMA buffer, the size of the transfer, and the *WDC_DMA* flags. **dma_start**() is called just after issuing a DMA command to the IDE device. The arguments are, respectively: *dma_arg*, the channel number, the drive number on this channel, and the *WDC_DMA* flags. **dma_finish**() is called once the transfer is complete. The arguments are, respectively: *dma_arg*, the channel number, the drive number on this channel, and the *WDC_DMA* flags. *WDC_DMA_READ* indicates the direction of the data transfer, and *WDC_DMA_POLL* indicates if the transfer will use (or used) interrupts.

The *WDC_CAPABILITY_MODE* flag means that the bus front-end can program the PIO and DMA modes, so **wdc** needs to provide back the supported modes for each drive, and set the drives modes. The *pio_mode* and *dma_mode* needs to be set to the highest PIO and DMA mode supported. If *WDC_CAPABILITY_UDMA* is set, then *dma_mode* must be set to the highest Ultra-DMA mode supported. If *WDC_CAPABILITY_MODE* is not set, **wdc** will not attempt to change the current drive's settings, assuming the host's firmware has done it right.

The *WDC_CAPABILITY_HWLOCK* flag is set for controllers needing hardware looking before accessing the I/O ports. If this flag is set, the bus front-end needs to provide the **claim_hw**() and **free_hw**() functions. **claim_hw**() will be called when the driver wants to access the controller ports. The second parameter is set to 1 when it is possible to sleep waiting for the lock, 0 otherwise. It should return 1 when access has been granted, 0 otherwise. When access has not been granted and sleep is not allowed, the bus front-end shall call **wdcrestart**() with the first argument passed to **claim_hw**() as argument. This arguments will also be the one passed to **free_hw**(). This function is called once the transfer is complete, so that the lock can be released.

Accesses to the data port are done by using the bus_space stream functions, unless the *WDC_CAPABILITY_ATA_NOSTREAM* or *WDC_CAPABILITY_ATAPI_NOSTREAM* flags are set. This should not be used, unless the data bus is not wired properly (which seems common on big-endian systems), and byte-order needs to be preserved for compatibility with the host's firmware. Also note that the IDE bus is a little-endian bus, so the bus_space functions used for the bus_space tag passed in the *channel_softc* have to do the appropriate byte-swapping for big-endian systems.

WDC_CAPABILITY_NO_EXTRA_RESETS avoid the controller reset at the end of the disks probe. This reset is needed for some controllers, but causes problems with some others.

WDC_CAPABILITY_NOIRQ tells the driver that this controller doesn't have its interrupt lines wired up usefully, so it should always use polled transfers.

The bus front-end needs to fill in the following elements of *channel_softc*:

channel	The channel number on the controller
wdc	A pointer to the controller's wdc_softc
cmd_iot, cmd_ioh	Bus-space tag and handle for access to the command block registers (which includes the 16-bit data port)
ctl_iot, ctl_ioh	Bus-space tag and handle for access to the control block registers
ch_queue	A pointer to a *struct channel_queue*. This will hold the queues of outstanding commands for this controller.

The following elements are optional:

data32iot, data32ioh
: Bus-space tag and handle for 32-bit data accesses. Only needed if *WDC_CAPABILITY_DATA32* is set in the controller's *wdc_softc*.

ch_queue can point to a common *struct channel_queue* if the controller doesn't support concurrent access to its different channels. If all channels are independent, it is recommended that each channel has its own *ch_queue* (for better performance).

The bus-specific front-end can use the **wdcprobe**() function, with a properly initialised *struct channel_softc* as argument (*wdc* can be set to NULL. This allows **wdcprobe**() to be easily used in bus front-end probe functions). This function will return an integer where bit 0 will be set if the master device has been found, and 1 if the slave device has been found.

The bus-specific attach function has to call **wdcattach**() for each channel, with a pointer to a properly initialised *channel softc* as argument. This will probe devices attached to the IDE channel and attach them. Once this function returns, the *ch_drive* array of the *channel_softc* will contain the drive's capabilities. This can be used to properly initialise the controller's mode, or disable a channel without drives.

The elements of interest in *ata_drive_datas* for a bus front-end are:

drive
: The drive number

drive_flags
: Flags indicating the drive capabilities. A null *drive_flags* indicate either that no drive is here, or that no driver was found for this device.

PIO_mode, DMA_mode, UDMA_mode
: the highest supported modes for this drive compatible with the controller's capabilities. Needs to be reset to the mode to use by the drive, if known.

drv_softc
: A pointer to the drive's softc. Can be used to print the drive's name.

drive_flags handles the following flags:

DRIVE_ATA, DRIVE_ATAPI
: Gives the drive type, if any. The shortcut DRIVE can be used to just test the presence/absence of a drive.

DRIVE_CAP32
: This drive works with 32-bit data I/O.

DRIVE_DMA
: This drive supports DMA.

DRIVE_UDMA
: This drive supports Ultra-DMA.

DRIVE_MODE
: This drive properly reported its PIO and DMA mode.

Once the controller has been initialised, it has to reset the *DRIVE_DMA* and *DRIVE_UDMA*, as well as the values of *PIO_mode*, *DMA_mode* and *UDMA_mode* if the modes to be used are not highest ones supported by the drive.

SEE ALSO

wdc(4), bus_space(9)

CODE REFERENCES

The wdc core functions are implemented in `sys/dev/ic/wdc.c`. Low-level ATA and ATAPI support is provided by `sys/dev/ata_wdc.c` and `sys/dev/scsipi/atapi_wdc.c` respectively.

An example of a simple bus front-end can be found in `sys/dev/isapnp/wdc_isapnp.c`. A more complex one, with multiple channels and bus-master DMA support is `sys/dev/pci/pciide.c`. `sys/arch/atari/dev/wdc_mb.c` makes use of hardware locking, and also provides an example of bus-front end for a big-endian system, which needs byte-swapping bus_space functions.

NAME

workqueue — simple do-it-in-thread-context framework

DESCRIPTION

The **workqueue** utility routines are provided to defer work which is needed to be processed in a thread context.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing the **workqueue** subsystem can be found. All pathnames are relative to `/usr/src`.

The **workqueue** subsystem is implemented within the file `sys/kern/subr_workqueue.c`.

SEE ALSO

`intro`(9), `workqueue_create`(9), `workqueue_destroy`(9), `workqueue_enqueue`(9)

NAME

workqueue_create — Create a workqueue

SYNOPSIS

```
#include <sys/workqueue.h>

int
workqueue_create(struct workqueue **wqp, const char *name,
    void (*func)(struct work *, void *), void *arg, pri_t prio, int ipl,
    int flags);
```

DESCRIPTION

workqueue_create() creates a workqueue. It takes the following arguments.

wqp Specify where to store the created workqueue.

name The name of the workqueue.

func The function to be called for each *work*.

arg An argument to be passed as a second argument of *func*.

prio The process priority to be used when sleeping to wait requests.

ipl The highest IPL at which this workqueue is used.

flags The value of 0 indicates a standard create operation, however the following flags may be bitwise ORed together:

WQ_MPSAFE Specifies that the workqueue is multiprocessor safe and does its own locking, otherwise the kernel lock will be held while work will be processed.

WQ_PERCPU Specifies that the workqueue should have a separate queue for each CPU, thus the work could be enqueued on concrete CPUs.

RETURN VALUES

workqueue_create() returns 0 on success. Otherwise, it returns an errno(2).

SEE ALSO

errno(2), condvar(9), intro(9), workqueue(9)

NAME

workqueue_destroy — Destroy a workqueue

SYNOPSIS

```
#include <sys/workqueue.h>

void
workqueue_destroy(struct workqueue *wq);
```

DESCRIPTION

workqueue_destroy() destroys a workqueue and frees associated resources. The caller should ensure that the workqueue has no work enqueued beforehand.

SEE ALSO

intro(9), workqueue(9)

NAME

workqueue_enqueue — Enqueue a work for later processing

SYNOPSIS

```
#include <sys/workqueue.h>

void
workqueue_enqueue(struct workqueue *wq, struct work *wk,
    struct cpu_info *ci);
```

DESCRIPTION

Enqueue the work *wk* into the workqueue *wq*.

If the WQ_PERCPU flag was set on workqueue creation, the *ci* argument may be used to specify the CPU on which the work should be enqueued. Also it may be NULL, then work will be enqueued on the current CPU. If WQ_PERCPU flag was not set, *ci* must be NULL.

The enqueued work will be processed in a thread context. A work must not be enqueued again until the callback is called by the workqueue(9) framework.

SEE ALSO

intro(9), workqueue(9)

NAME

wscons — machine-independent console support

DESCRIPTION

The **wscons** driver provides a machine-independent framework for workstation consoles. It consists of several cooperating modules:

- display adapters (see wsdisplay(9))
- keyboards (see wskbd(9))
- pointers and mice (see wsmouse(9))
- input event multiplexor
- font handling (see wsfont(9))
- terminal emulation (see wsdisplay(9))

The wscons framework replaces the old rcons workstation framework and the various machine-dependent console implementations.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent wscons subsystem can be found. All pathnames are relative to /usr/src.

The wscons subsystem is implemented within the directory sys/dev/wscons.

SEE ALSO

wscons(4), cons(9), driver(9), intro(9), wsdisplay(9), wsfont(9), wskbd(9), wsmouse(9)

NAME

wsdisplay, **wsdisplay_switchtoconsole**, **wsdisplay_cnattach**, **wsdisplaydevprint**, **wsemuldisplaydevprint** — wscons display support

SYNOPSIS

```
#include <dev/wscons/wsconsio.h>
#include <dev/wscons/wsdisplayvar.h>
#include <dev/wscons/wsemulvar.h>
#include <dev/wscons/wsemul_vt100var.h>

void
wsdisplay_switchtoconsole();

void
wsdisplay_cnattach(const struct wsscreen_descr *type, void *cookie,
    int ccol, int crow, long defattr);

void
wsemul_xxx_cnattach(const struct wsscreen_descr *type, void *cookie,
    int ccol, int crow, long defattr);

int
wsdisplaydevprint(void *aux, const char *pnp);

int
wsemuldisplaydevprint(void * aux, const char *pnp);
```

DESCRIPTION

The **wsdisplay** module is a component of the wscons(9) framework to provide machine-independent display support. Most of the support is provided by the wsdisplay(4) device driver, which must be a child of the hardware device driver.

The wscons display interface is complicated by the fact that there are two different interfaces. The first interface corresponds to the simple bit-mapped display which doesn't provide terminal-emulation and console facilities. The second interface provides machine-independent terminal emulation for displays that can support glass-tty terminal emulations. These are character-oriented displays, with row and column numbers starting at zero in the upper left hand corner of the screen. Display drivers which cannot emulate terminals use the first interface. In most cases, the low-level hardware driver can use the rasops(9) interface to provide enough support to allow glass-tty terminal emulation. If the display is not the console, terminal emulation does not make sense and the display operates using the bit-mapped interface.

The wscons framework allows concurrent displays to be active. It also provides support for multiple screens for each display and therefore allows a virtual terminal on each screen. Multiple terminal emulations and fonts can be active at the same time allowing different emulations and fonts for each screen.

Font manipulation facilities for the terminal emulation interface are available through the wsfont(9) module.

DATA TYPES

Display drivers providing support for wscons displays will make use of the following data types:

struct wsdisplay_accessops

A structure used to specify the display access functions invoked by userland program which require direct device access, such as X11. All displays must provide this structure and pass it to the wsdisplay(4) child device. It has the following members:

```
int     (*ioctl)(void *v, void *vs, u_long cmd,
                void *data, int flag, struct lwp *l);
paddr_t (*mmap)(void *v, void *vs, off_t off, int prot);
int     (*alloc_screen)(void *,
                const struct wsscreen_descr *, void **,
                int *, int *, long *);
void    (*free_screen)(void *, void *);
int     (*show_screen)(void *, void *, int,
                void (*)(), void *);
int     (*load_font)(void *, void *,
                struct wsdisplay_font *);
void    (*pollc)(void *, int);
void    (*scroll)(void *, void *, int);
```

The *ioctl* member defines the function to be called to perform display-specific ioctl calls. The *mmap* member defines the function for mapping a part of the display device into user address space. The *alloc_screen* member defines a function for allocating a new screen which can be used as a virtual terminal. The *free_screen* member defines a function for de-allocating a screen. The *show_screen* member defines a function for mapping a screen onto the physical display. This function is used for between switching screens. The *load_font* member defines a function for loading a new font into the display. The *pollc* member defines a function for polling the console. The *scroll* member defines a function for scrolling the contents of the display.

There is a *void ** cookie provided by the display driver associated with these functions, which is passed to them when they are invoked.

The *void *vs* cookie, passed to **ioctl**() and **mmap**(), points to the virtual screen on which these operations were executed.

struct wsdisplaydev_attach_args

A structure used to attach the wsdisplay(4) child device for the simple bit-mapped interface. If the full terminal-emulation interface is to be used, then *struct wsemuldisplaydev_attach_args* should be used instead. It has the following members:

```
const struct wsdisplay_accessops *accessops;
void *accesscookie;
```

struct wsemuldisplaydev_attach_args

A structure used to attach the wsdisplay(4) child device for the full terminal emulation interface. If the simple bit-mapped interface is to be used, then *struct wsdisplaydev_attach_args* should be used instead. It has the following members:

```
int console;
const struct wsscreen_list *scrdata;
const struct wsdisplay_accessops *accessops;
void *accesscookie;
```

struct wsdisplay_emulops

A structure used to specify the display emulation functions. All displays intending to provide terminal emulation must provide this structure and pass it to the wsdisplay(4) child device. It has the following members:

```
void    (*cursor)(void *c, int on, int row, int col);
int     (*mapchar)(void *, int, unsigned int *);
void    (*putchar)(void *c, int row, int col,
```

```
                u_int uc, long attr);
        void    (*copycols)(void *c, int row, int srccol,
                int dstcol, int ncols);
        void    (*erasecols)(void *c, int row, int startcol,
                int ncols, long);
        void    (*copyrows)(void *c, int srcrow, int dstrow,
                int nrows);
        void    (*eraserows)(void *c, int row, int nrows, long);
        int     (*allocattr)(void *c, int fg, int bg, int flags,
                long *);
        void    (*replaceattr)(void *c, long oldattr,
                long newattr);
```

The *cursor* member defines a function for painting (or unpainting, depending on the *on* parameter) the cursor at the specified position. The *mapchar* member defines a function for changing the character mapped at a given position in the character table. The *putchar* member defines a function for writing a character on the screen, given its position and attribute. The *copycols* member defines a function for copying a set of columns within the same line. The *erasecols* member defines a function for clearing a set of columns in a line, filling the space with the given attribute. The *copyrows* member defines a function for copying a set of complete rows. The *eraserows* member defines a function for clearing a set of complete rows, filling the space with the given attribute. The *allocattr* member defines a function for converting an attribute specification given by its foreground color, background color and flags, to the internal representation used by the underlying graphics driver. The *replaceattr* member defines a function for replacing an attribute by another one across the whole visible part of the screen; this function is optional.

There is a *void ** cookie provided by the display driver associated with these functions, which is passed to them when they are invoked.

struct wsscreen_descr

A structure passed to wscons by the display driver to describe a screen. All displays which can operate as a console must provide this structure and pass it to the wsdisplay(4) child device. It contains the following members:

```
char *name;
int ncols, nrows;
const struct wsdisplay_emulops *textops;
int fontwidth, fontheight;
int capabilities;
```

The *capabilities* member is a set of flags describing the screen capabilities. It can contain the following flags:

WSSCREEN_WSCOLORS	minimal color capability
WSSCREEN_REVERSE	can display reversed
WSSCREEN_HILIT	can highlight (however)
WSSCREEN_BLINK	can blink
WSSCREEN_UNDERLINE	can underline

struct wsscreen_list

A structure passed to wscons by the display driver to tell about its capabilities. It contains the following members:

```
int nscreens;
const struct wsscreen_descr **screens;
```

struct wscons_syncops

A structure passed to wscons by the display driver describing the interface for external screen switching/process synchronization. This structure is optional and only required by displays operating with terminal emulation and intending to support multiple screens. It contains the following members:

```
int     (*detach)(void *, int, void (*)(), void *);
int     (*attach)(void *, int, void (*)(), void *);
int     (*check)(void *);
void    (*destroy)(void *);
```

FUNCTIONS

wsdisplay_switchtoconsole()

Switch the console display to its first screen.

wsdisplay_cnattach(*type*, *cookie*, *ccol*, *crow*, *defattr*)

Attach this display as the console input by specifying the number of columns *ccol* and number of rows *crows*. The argument *defattr* specifies the default attribute (color) for the console.

wsemul_xxx_cnattach(*type*, *cookie*, *ccol*, *crow*, *defattr*)

Attach this display as the console with terminal emulation described by the *xxx* and specifying the number of columns *ccol* and number of rows *crows*. The argument *defattr* specifies the default attribute (color) for the console. Different terminal emulations can be active at the same time on one display.

wsdisplaydevprint(*aux*, *pnp*)

The default wsdisplay printing routine used by **config_found**(). (see autoconf(9)).

wsemuldisplaydevprint(*aux*, *pnp*)

The default wsemul printing routine used by **config_found**(). (see autoconf(9)).

AUTOCONFIGURATION

Display drivers which want to use the wsdisplay module must be a parent to the wsdisplay(4) device and provide an attachment interface. To attach the wsdisplay(4) device, the display driver must allocate and populate a *wsdisplaydev_attach_args* structure with the supported operations and callbacks and call **config_found**() to perform the attach (see autoconf(9)).

Display drivers which want to use the wscons terminal emulation module must be a parent to the wsdisplay(4) device and provide a *wsemuldisplaydev_attach_args* structure instead of the standard *wsdisplaydev_attach_args* to **config_found**() to perform the attach. If the display is not the console the attachment is the same as wsdisplaydev_attach_args.

OPERATION

If the display belongs to the system console, it must describe the default screen by invoking **wsdisplay_cnattach**() at console attach time.

All display manipulation is performed by the wscons interface by using the callbacks defined in the *wsdisplay_accessops* structure. The **ioctl**() function is called by the wscons interface to perform display-specific ioctl operations (see ioctl(2)). The argument *cmd* to the **ioctl**() function specifies the specific command to perform using the data *data*. Valid commands are listed in sys/dev/wscons/wsconsio.h and documented in wsdisplay(4). Operations for terminal emulation are performed using the callbacks defined in the *wsdisplay_emulops* structure.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent wscons subsystem can be found. All pathnames are relative to `/usr/src`.

The wscons subsystem is implemented within the directory `sys/dev/wscons`. The **`wsdisplay`** module itself is implemented within the file `sys/dev/wscons/wsdisplay.c`. The terminal emulation support is implemented within the files `sys/dev/wscons/wsemul_*`. `ioctl`(2) operations are listed in `sys/dev/wscons/wsconsio.h`.

SEE ALSO

`ioctl`(2), `wsdisplay`(4), `autoconf`(9), `driver`(9), `intro`(9), `rasops`(9), `wsfont`(9), `wskbd`(9), `wsmouse`(9)

NAME

wsfont, **wsfont_init**, **wsfont_matches**, **wsfont_find**, **wsfont_add**, **wsfont_remove**, **wsfont_enum**, **wsfont_lock**, **wsfont_unlock**, **wsfont_getflg**, **wsfont_map_unichar** — wscons font support

SYNOPSIS

```
#include <dev/wscons/wsconsio.h>
#include <dev/wsfont/wsfont.h>

void
wsfont_init(void);

int
wsfont_matches(struct wsdisplay_font *font, const char *name, int width,
    int height, int stride);

int
wsfont_find(const char *name, int width, int height, int stride,
    int bitorder, int byteorder);

int
wsfont_add(struct wsdisplay_font *font, int copy);

int
wsfont_remove(int cookie);

void
wsfont_enum(void (*callback)(const char *, int, int, int));

int
wsfont_lock(int cookie, struct wsdisplay_font **ptr);

int
wsfont_unlock(int cookie);

int
wsfont_getflg(int cookie, int *flg, int *lc);

int
wsfont_map_unichar(struct wsdisplay_font *font, int c);
```

DESCRIPTION

The **wsfont** module is a component of the wscons(9) framework to provide access to display fonts. Fonts may be loaded dynamically into the kernel or included statically in the kernel at compile time. Display drivers which emulate a glass-tty console on a bit-mapped display can add, remove and find fonts for use by device-dependent blitter operations.

The primary data type for manipulating fonts is the *wsdisplay_font* structure in dev/wscons/wsconsio.h:

```
struct wsdisplay_font {
        char *name;             /* font name */
        int firstchar;
        int numchars;           /* size of font table */
        int encoding;           /* font encoding
        u_int fontwidth;        /* character width */
        u_int fontheight;       /* character width */
        u_int stride;
```

```
        int bitorder;
        int byteorder;
        void *data;            /* pointer to font table */
};
```

The maximum font table size is *WSDISPLAY_MAXFONTSZ*.

The **wsfont** framework supports fonts with the following encodings:

WSDISPLAY_FONTENC_ISO
: ISO-encoded fonts.

WSDISPLAY_FONTENC_IBM
: IBM-encoded fonts commonly available for IBM CGA, EGA and VGA display adapters.

WSDISPLAY_FONTENC_PCVT
: PCVT-encoding fonts distributed as part of the old PCVT terminal emulation driver.

WSDISPLAY_FONTENC_ISO7
: ISO-encoded Greek fonts.

WSDISPLAY_FONTENC_ISO2
: ISO-encoded East European fonts.

FUNCTIONS

wsfont_init(*void*)
: Initialise the font list with the built-in fonts.

wsfont_matches(*font*, *name*, *width*, *height*, *stride*)
: Matches the font *font* with the specifications *name*, *width*, *height* and *stride*. Return zero if not matched and non-zero if matched.

wsfont_find(*name*, *width*, *height*, *stride*, *bitorder*, *byteorder*)
: Find the font called *name* from the fonts loaded into the kernel. The font aspect is specified by *width*, *height*, and *stride*. If **wsfont_find**() is called with any of the parameters as 0, it indicates that we don't care about that aspect of the font. If the font is found a (nonnegative-valued) cookie is returned which can used with the other functions.

  The *bitorder* and *byteorder* arguments are the bit order and byte order required. Valid values are:

  WSDISPLAY_FONTORDER_KNOWN
  : The font is in known ordered format and doesn't need converting.

  WSDISPLAY_FONTORDER_L2R
  : The font is ordered left to right.

  WSDISPLAY_FONTORDER_R2L
  : The font is ordered right to left.

  When more flexibility is required, **wsfont_enum**() should be used.

wsfont_add(*font*, *copy*)
: Add a font *font* to the font list. If the *copy* argument is non-zero, then the font is physically copied, otherwise a reference to the original font is made.

wsfont_remove(*cookie*)
: Remove the font specified by *cookie* from the font list. The value of cookie was returned by **wsfont_add**().

wsfont_enum(*callback*)
Enumerate the list of fonts. For each font in the font list, the *callback* function argument is called with the arguments specifying the font name, width, height and stride.

wsfont_lock(*cookie*, *ptr*)
Lock access to the font specified by *cookie* so that it cannot be unloaded from the kernel while is being used. If the bit or byte order of the font to be locked differs from what has been requested with **wsfont_find**() then the glyph data will be modified to match. At this point it may be necessary for **wsfont_lock**() to make a copy of the font data; this action is transparent to the caller. A later call to **wsfont_unlock**() will free resources used by temporary copies.

The address of the wsdisplay_font pointer for the specified font is return in the *ptr* argument.

wsfont_lock() returns zero on success, or an error code on failure.

wsfont_unlock(*cookie*)
Unlock the font specified by *cookie*. Returns zero on success, or an error code on failure.

wsfont_map_unichar(*font*, *c*)
Remap the unicode character *c* to glyph for font *font*. Returns the glyph or success or -1 on error.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent wsfont subsystem can be found. All pathnames are relative to `/usr/src`.

The wscons subsystem is implemented within the directory `sys/dev/wscons`. The wsfont subsystem itself is implemented within the file `sys/dev/wsfont/wsfont.c`.

SEE ALSO

`wsfont`(4), `wsfontload`(8), `autoconf`(9), `driver`(9), `intro`(9), `wscons`(9), `wsdisplay`(9)

NAME

wskbd, **wskbd_input**, **wskbd_rawinput**, **wskbd_cnattach**, **wskbd_cndetach**, **wskbddevprint** — wscons keyboard support

SYNOPSIS

```
#include <dev/wscons/wsconsio.h>
#include <dev/wscons/wskbdvar.h>
#include <dev/wscons/wsksymdef.h>
#include <dev/wscons/wsksymvar.h>

void
wskbd_input(struct device *kbddev, u_int type, int value);

void
wskbd_rawinput(struct device *kbddev, u_char *buf, int len);

void
wskbd_cnattach(const struct wskbd_consops *consops, void *conscookie,
    const struct wskbd_mapdata *mapdata);

void
wskbd_cndetach();

int
wskbddevprint(void *aux, const char *pnp);
```

DESCRIPTION

The **wskbd** module is a component of the wscons(9) framework to provide machine-independent keyboard support. Most of the support is provided by the wskbd(4) device driver, which must be a child of the hardware device driver.

DATA TYPES

Keyboard drivers providing support for wscons keyboards will make use of the following data types:

kbd_t An opaque type describing keyboard properties.

keysym_t

The wscons keyboard-independent symbolic representation of the keypress.

struct wskbd_accessops

A structure used to specify the keyboard access functions. All keyboards must provide this structure and pass it to the wskbd(4) child device. It has the following members:

```
int     (*enable)(void *, int);
void    (*set_leds)(void *, int);
int     (*ioctl)(void *v, u_long cmd, void *data,
                 int flag, struct lwp *l);
```

The *enable* member defines the function to be called to enable keypress passing to wscons. The *set_leds* member defined the function to be called to set the LEDs on the keyboard. The *ioctl* member defines the function to be called to perform keyboard-specific ioctl calls.

There is a *void * * cookie provided by the keyboard driver associated with these functions, which is passed to them when they are invoked.

struct wskbd_consops

A structure used to specify the keyboard console operations. All keyboards which can operate as a console must provide this structure and pass it to the wskbd(4) child device. If the keyboard

cannot be a console, it is not necessary to specify this structure. It has the following members:

```
void    (*getc)(void *, u_int *, int *);
void    (*pollc)(void *, int);
void    (*bell)(void *, u_int, u_int, u_int);
```

There is a *void ** cookie provided by the keyboard driver associated with these functions, which is passed to them when they are invoked.

struct wscons_keydesc

A structure used to describe a keyboard mapping table to convert keyboard-specific keycodes to wscons keysyms. It has the following members:

```
kbd_t   name;           /* name of this map */
kbd_t   base;           /* map this one is based on */
int     map_size;       /* size of map */
const   keysym_t *map;  /* the map itself */
```

struct wskbd_mapdata

A structure used to describe the keyboard layout and operation to interpret the keyboard layout. it contains the following members:

```
const struct wscons_keydesc *keydesc;
kbd_t layout;
```

struct wskbddev_attach_args

A structure used to attach the wskbd(4) child device. It has the following members:

```
int console;
const struct wskbd_mapdata *keymap;
const struct wskbd_accessops *accessops;
void *accesscookie;
```

Keymaps

Keymaps are a dense stream of *keysym_t*. A declaration has the following fields:

pos [*cmd*] *normal* [*shift*] [*altgr*] [*shift-altgr*]

The fields have the following meanings:

pos	Always specified as KC(*pos*) and starts the description of key *pos*.
cmd	If the command modifier (KS_Cmd_XXX) is active, the optional command *cmd* is invoked.
normal	The keysym if no modifiers are active.
shift	The keysym if the shift modifier is active.
altgr	The keysym if the alt-gr modifier is active.
shift-altgr	The keysym if the shift-alt-gr modifier is active.

If the keysym after *pos* is not KS_Cmd_XXX, then *cmd* is empty. The *shift*, *altgr* and *shift-altgr* fields are determined from previous fields if they are not specified. Therefore, there are four valid keysym declarations:

pos [*cmd*] *normal*

pos [*cmd*] *normal shift*

pos [*cmd*] *normal shift altgr*

pos [*cmd*] *normal shift altgr shift-altgr*

FUNCTIONS

wskbd_input(*kbddev*, *type*, *value*)

Pass the keypress of value *value* and type *type* to wscons keyboard driver. Valid values of *type* are:

WSCONS_EVENT_KEY_UP
: Key released.

WSCONS_EVENT_KEY_DOWN
: Key pressed.

wskbd_rawinput(*kbddev*, *buf*, *len*)

Pass the raw keypress in the buffer *buf* to the wscons keyboard driver. The buffer is *len* bytes long. This function should only be called if the kernel option *WSDISPLAY_COMPAT_RAWKBD* is enabled.

wskbd_cnattach(*consops*, *conscookie*, *mapdata*)

Attach this keyboard as the console input by specifying the console operations *consops* and the keyboard mapping table information in *mapdata*. The functions specified in *consops* will be called with *conscookie* as the first argument.

wskbd_cndetach()

Detach this keyboard as the console input.

wskbddevprint(*aux*, *pnp*)

The default wskbd printing routine used by **config_found**(). (see autoconf(9)).

AUTOCONFIGURATION

Keyboard drivers which want to use the wskbd module must be a parent to the wskbd(4) device and provide an attachment interface. To attach the wskbd(4) device, the keyboard driver must allocate and populate a *wskbddev_attach_args* structure with the supported operations and callbacks and call **config_found**() to perform the attach (see autoconf(9)). The *keymap* member points to the *wskbd_mapdata* structure which describes the keycode mapping operations. The *accessops* member points to the *wskbd_accessops* structure which describes the keyboard access operations. The *console* member is a boolean to indicate to wscons whether this keyboard will be used for console input.

OPERATION

If the keyboard belongs to the system console, it must register the *wskbd_consops* structure specifying the console operations via **wskbd_cnattach**() at console attach time.

When a keypress arrives from the keyboard, the keyboard driver must perform any necessary character decoding to wscons events and pass the events to wscons via **wskbd_input**(). If the kernel is compiled with the option *WSDISPLAY_COMPAT_RAWKBD*, then the keyboard driver must also pass the raw keyboard data to wscons via **wskbd_rawinput**().

The wscons framework calls back into the hardware driver by invoking the functions that are specified in the *accessops* structure. The **enable**() and **set_leds**() functions are relatively simple and self-explanatory. The **ioctl**() function is called by the wscons interface to perform keyboard-specific ioctl operations (see ioctl(2)). The argument *cmd* to the **ioctl**() function specifies the specific command to perform using the data *data*. Valid commands are listed in sys/dev/wscons/wsconsio.h.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent wscons subsystem can be found. All pathnames are relative to `/usr/src`.

The wscons subsystem is implemented within the directory `sys/dev/wscons`. The **wskbd** module itself is implement within the files `sys/dev/wscons/wskbd.c` and `sys/dev/wscons/wskbdutil.c`. `ioctl`(2) operations are listed in `sys/dev/wscons/wsconsio.h`.

SEE ALSO

`ioctl`(2), `autoconf`(9), `driver`(9), `intro`(9), `wsdisplay`(9), `wsmouse`(9)

NAME

wsmouse, **wsmouse_input**, **wsmousedevprint** — wscons mouse support

SYNOPSIS

```
#include <dev/wscons/wsconsio.h>
#include <dev/wscons/wsmousevar.h>

void
wsmouse_input(struct device *msdev, u_int btns, int x, int y, int z, int w,
    u_int flags);

int
wsmousedevprint(void *aux, const char *pnp);
```

DESCRIPTION

The **wsmouse** module is a component of the wscons(9) framework to provide machine-independent mouse support. Most of the support is provided by the wsmouse(4) device driver, which must be a child of the hardware device driver.

DATA TYPES

Mouse drivers providing support for wscons pointer devices will make use of the following data types:

struct wsmouse_accessops

A structure used to specify the mouse access functions. All pointer devices must provide this structure and pass it to the wsmouse(4) child device. It has the following members:

```
int     (*enable)(void *);
int     (*ioctl)(void *v, u_long cmd, void *data,
                 int flag, struct lwp *l);
void    (*disable)(void *);
```

The *enable* member defines the function to be called to enable monitoring pointer movements and passing these events to wscons. The *disable* member defines the function to disable movement events. The *ioctl* member defines the function to be called to perform mouse-specific ioctl calls.

There is a *void * * cookie provided by the mouse driver associated with these functions, which is passed to them when they are invoked.

struct wsmousedev_attach_args

A structure used to attach the wsmouse(4) child device. It has the following members:

```
const struct wsmouse_accessops *accessops;
void *accesscookie;
```

FUNCTIONS

wsmouse_input(*msdev*, *btns*, *x*, *y*, *z*, *w*, *flags*)

Callback from the mouse driver to the wsmouse interface driver. Arguments are as follows:

msdev This is the *struct device* pointer passed from **config_found**() on attaching the child wsmouse(4) to specify the mouse device.

btns This specifies the current button status. Bits for pressed buttons (which will cause the WSCONS_EVENT_MOUSE_DOWN event on wsmouse(4) device) should be set, and bits for released buttons (which will cause the WSCONS_EVENT_MOUSE_UP event) should be zero. The left most button state should be in LSB, i.e. for typical three button mouse, the left button is 0x01, the middle button is 0x02, and the right button is 0x04.

x Absolute or relative X-axis value to specify the pointer coordinate. Rightward (moving the mouse right) is positive.

y Absolute or relative Y-axis value to specify the pointer coordinate. Upward (moving the mouse forward) is positive. Note that this aspect is opposite from the one used in the X server dix layer.

z Absolute or relative Z-axis value to specify the pointer coordinate. Usually this axis is used for the wheel. Downward (turning the wheel backward) is positive.

w Absolute or relative W-axis value to specify the pointer coordinate. Usually this axis would be used for the horizontal component of the wheel.

flags This argument specifies whether the pointer device and the measurement of the *x*, *y*, *z*, and *w* axes is in relative or absolute mode. Valid values for *flags* are:

WSMOUSE_INPUT_DELTA
Relative mode.

WSMOUSE_INPUT_ABSOLUTE_X
Absolute mode in *x* axis.

WSMOUSE_INPUT_ABSOLUTE_Y
Absolute mode in *y* axis.

WSMOUSE_INPUT_ABSOLUTE_Z
Absolute mode in *z* axis.

WSMOUSE_INPUT_ABSOLUTE_W
Absolute mode in *w* axis.

wsmousedevprint(*aux*, *pnp*)
The default wsmouse printing routine used by **config_found**(). (see autoconf(9)).

AUTOCONFIGURATION

Mouse drivers which want to use the wsmouse module must be a parent to the wsmouse(4) device and provide an attachment interface. To attach the wsmouse(4) device, the mouse driver must allocate and populate a *wsmousedev_attach_args* structure with the supported operations and callbacks and call **config_found**() to perform the attach (see autoconf(9)).

OPERATION

When a mouse-movement event is received, the device driver must perform any necessary movement decoding to wscons events and pass the events to wscons via **wsmouse_input**().

The wscons framework calls back into the hardware driver by invoking the functions that are specified in the *accessops* structure. The **enable**() and **disable**() functions are relatively simple and self-explanatory. The **ioctl**() function is called by the wscons interface to perform mouse-specific ioctl operations (see ioctl(2)). The argument *cmd* to the **ioctl**() function specifies the specific command to perform using the data *data*. Valid commands are listed in sys/dev/wscons/wsconsio.h.

CODE REFERENCES

This section describes places within the NetBSD source tree where actual code implementing or using the machine-independent wscons subsystem can be found. All pathnames are relative to /usr/src.

The wscons subsystem is implemented within the directory sys/dev/wscons. The **wsmouse** module itself is implement within the file sys/dev/wscons/wsmouse.c. ioctl(2) operations are listed in sys/dev/wscons/wsconsio.h.

SEE ALSO

`ioctl`(2), `pms`(4), `wscons`(4), `wsmouse`(4), `autoconf`(9), `driver`(9), `intro`(9), `wscons`(9), `wsdisplay`(9), `wskbd`(9)