2 Kernel Reference Pages
This chapter contains reference pages for the kernel support routines commonly used by I/O drivers.
Functions and Structures
NAME

`alloc_spinlock(KER2)` – Allocate and initialize a spinlock resource.

SYNOPSIS

```c
#include <sys/spinlock.h>

lock_t *alloc_spinlock (unsigned int order, char *name);
```

PARAMETERS

- **order**: Lock order.
- **name**: Spinlock name.

DESCRIPTION

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread's current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The `alloc_spinlock()` kernel function allocates and initializes a spinlock resource. The caller is responsible for deallocating the spinlock resource when it is no longer needed. See `dealloc_spinlock(KER2)`.

The `order` parameter is the lock order of the spinlock. The `order` must be chosen such that deadlocks with other spinlocks are avoided. To avoid deadlocks, the spinlock to be acquired must have a lock order greater than that of any spinlock currently held by the processor. See spinlock orders in `<sys/semglobal.h>`.

The `name` parameter points to a character string containing the name of the spinlock.

RETURN VALUES

`alloc_spinlock()` returns a pointer to a spinlock resource.
CONSTRANTS

Must not be called in an interrupt context.
Must not be called while holding a spinlock.

EXAMPLE

/*
 * Choose a lock order for my driver that is lower than
 * any spinlock used by a service that the driver may
 * call. In <sys/semglobal.h>, SPL_LOCK_ORDER is
 * defined with a sufficiently low lock order value.
 */
#define MYDRV_LOCK_ORDER SPL_LOCK_ORDER

lock_t * mydrv_lock;

/*
 * Allocate a spinlock resource. alloc_spinlock() does
 * not return until it has successfully allocated the
 * memory for the spinlock and initialized the spinlock.
 */
mydrv_lock = alloc_spinlock(MYDRV_LOCK_ORDER,
                           "mydrv spinlock");

SEE ALSO

cspinlock(KER2), dealloc_spinlock(KER2),
owns_spinlock(KER2), spinlock(KER2), spinunlock(KER2)
NAME

*b_cpsema* (KER2) – Conditionally acquire (lock) a beta semaphore.

SYNOPSIS

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

int b_cpsema (struct b_sema * sema);
```

PARAMETERS

*sema* Pointer to a *b_sema* structure.

DESCRIPTION

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

The *b_cpsema()* kernel function attempts to conditionally acquire (lock) a beta semaphore pointed to by *sema*. The calling thread is not blocked if the beta semaphore is currently owned.

RETURN VALUES

1 Acquired (locked) the beta semaphore
0 The beta semaphore is currently owned.

CONSTRAINTS

Must not be called in an interrupt context.
Must not be called while holding a spinlock with lock order \( \geq \) SEMAPHORE_LOCK_ORDER.
EXAMPLE

static b_sema_t mydrv_sema_1;
static b_sema_t mydrv_sema_2;
...

/*
 * Acquire a beta semaphore. This is the first of two
 * beta semaphores that will be needed, but we must
 * acquire mydrv_sema_2 before mydrv_sema_1 which is the
 * wrong lock order. We get away with doing this by
 * trying to conditionally acquire
 * mydrv_sema_1 later.
 */
b_psema(&mydrv_sema_2);
...

/*
 * Try to conditionally acquire another beta semaphore.
 * This violates the normal lock order (the previous
 * beta semaphore has a higher lock order), but this is
 * allowed since b_cpsema() does not block the thread.
 */
if (!b_cpsema(&mydrv_sema_1)) {
    /*
    * Failed to acquire the next beta semaphore.
    * As a sanity check, assert that we are not
    * the thread that owns the beta semaphore.
    */
    VASSERT(!b_owns_sema(&mydrv_sema_1));

    /*
    * Release mydrv_sema_2 to allow the other thread
    * that owns mydrv_sema_1 to make forward progress,
    * otherwise a deadlock condition will exist.
    */
    b_vsema(&mydrv_sema_2);

    /*
    * Now acquire the two beta semaphores in the correct
    * order to avoid a deadlock. We will probably block
    * here until the other thread releases mydrv_sema_1.
    */
    b_psema(&mydrv_sema_1);
    b_psema(&mydrv_sema_2);
b_cpsema (KER2)

/*
 * Check if data have changed between the time we released
 * mydrv_sema_2 and reacquired the beta semaphores.
 */

SEE ALSO

b_initsema(KER2), b_owns_sema(KER2), b_psema(KER2),
b_vsema(KER2)
NAME

\texttt{b\_initsema(KER2)} – Initialize a beta semaphore.

SYNOPSIS

\begin{verbatim}
#include <sys/sem\_beta.h>

void b\_initsema (struct b\_sema *sema, int val, int order, 
                 char *name);
\end{verbatim}

PARAMETERS

\begin{description}
\item \texttt{sema} \hspace{1cm} Pointer to a \texttt{b\_sema} structure.
\item \texttt{val} \hspace{1cm} Initial value of \texttt{sema}. Normally set to 1.
\item \texttt{order} \hspace{1cm} Lock order.
\item \texttt{name} \hspace{1cm} Beta semaphore name.
\end{description}

DESCRIPTION

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

The \texttt{b\_initsema()} kernel function initializes the beta semaphore pointed to by the \texttt{sema} parameter. The caller is responsible for allocating the kernel memory that instantiates the \texttt{b\_sema} structure.

The \texttt{val} parameter is normally set to the value 1, indicating the beta semaphore is initially not owned (unlocked). If \texttt{val} is set to 0, the beta semaphore is initialized as owned (locked).

The \texttt{order} parameter is the lock order of the beta semaphore. \texttt{order} must be a positive value and chosen so that deadlocks with other beta semaphores are avoided. To avoid deadlocks, the semaphore to be acquired must have a lock order greater than that of any semaphore currently held by the thread. See beta-class semaphore lock orders in \texttt{<sys/semglobal.h>}.
If the `SEMA_DEADLOCK_SAFE` flag is set (ORed with `order`), deadlock detection is disabled for the special case where beta semaphores have the same lock order values. The caller takes responsibility for ensuring it acquires semaphores of equal lock order in a deadlock-safe manner. The caller, however, is not allowed to acquire a beta semaphore with a lock order less than that of another beta semaphore currently held by the thread.

The `name` parameter points to a character string containing the name of the beta semaphore. The character string must not be an empty string.

**RETURN VALUES**

None

**CONSTRAINTS**

Must not be called in an interrupt context.

Must not be called while holding a spinlock with lock order $\geq$ `LOCK_INIT_LOCK_ORDER`.

**EXAMPLE**

```c
/*
 * Allocate static storage for the beta semaphore.
 */
static b_sema_t mydrv_sema;
...

/*
 * Initialize the beta semaphore. The lock order value
 * `REAL_DRV_SEMA_ORDER` is defined in `<sys/semglobal.h>`.
 */
b_initsema(&mydrv_sema, 1, REAL_DRV_SEMA_ORDER,
        "mydrv sema");
```

**SEE ALSO**

`b_cpsema(KER2), b_owns_sema(KER2), b_psema(KER2), b_vsema(KER2)`
NAME

*b_owns_sema* (KER2) – Test whether a beta semaphore is owned by the calling thread.

SYNOPSIS

```c
#include <sys/sem_beta.h>

int b_owns_sema (struct b_sema * sema);
```

PARAMETERS

- `sema` Pointer to a `b_sema` structure.

DESCRIPTION

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

The `b_owns_sema()` kernel function tests whether a beta semaphore, pointed to by `sema`, is owned (locked) by the calling thread.

RETURN VALUES

- 1 The calling thread owns the beta semaphore.
- 0 The current thread does not own the beta semaphore. It may be another thread, or not owned by any thread.

CONSTRAINTS

- Must not be called in an interrupt context.
EXAMPLE

static b_sema_t mydrv_sema;
...

int got_sema_here = 0;

/*
 * We may have already acquired the semaphore before getting
 * here. Be sure to check that we do not own the semaphore
 * before attempting to acquiring the semaphore.
 */
if (!b_owns_sema(&mydrv_sema)) {
    b_psema(&mydrv_sema);
    got_sema_here = 1;
}

/*
 * Do work under the protection of the semaphore.
 */
...

/*
 * Release the semaphore if it was acquired here.
 */
if (got_sema_here) {
    b_vsema(&mydrv_sema);
}

SEE ALSO

b_cpsema(KER2), b_initsema(KER2), b_psema(KER2), b_vsema(KER2)
NAME

*b_psema*(KER2) – Acquire (lock) a beta semaphore.

SYNOPSIS

```c
#include <sys/sem_beta.h>

void b_psema (struct b_sema *sema);
```

PARAMETERS

*sema* Pointer to a b_sema structure.

DESCRIPTION

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

The *b_psema()* kernel function attempts to acquire (lock) a beta semaphore pointed to by *sema*. The calling thread is blocked if the beta semaphore is currently owned.

RETURN VALUES

None

CONSTRAINTS

Must not be called in an interrupt context.
Must not be called while holding a spinlock.
EXAMPLE

static b_sema_t mydrv_sema;
...

/*
 * Acquire the beta semaphore for my driver.
 */
 b_psema(&mydrv_sema);

/*
 * Manipulate driver data protected by the beta semaphore.
 * Note: It is OK to sleep while holding a beta semaphore.
 */
...

/*
 * Release the beta semaphore when done.
 */
 b_vsema(&mydrv_sema);

SEE ALSO

b_cpsema(KER2), b_initsema(KER2), b_owns_sema(KER2),
b_vsema(KER2)
NAME

b_vsema(KER2) – Release (unlock) a beta semaphore.

SYNOPSIS

#include <sys/sem_beta.h>

void b_vsema (struct b_sema * sema);

PARAMETERS

sema Pointer to a b_sema structure.

DESCRIPTION

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

The b_vsema() kernel function releases (unlocks) the beta semaphore pointed to by sema.

RETURN VALUES

None.

CONSTRAINTS

Must not be called in an interrupt context.
EXAMPLE

```c
static b_sema_t mydrv_sema;
...

/*
 * Acquire the beta semaphore for my driver.
 */
b_psema(&mydrv_sema);

/*
 * Manipulate driver data protected by the beta semaphore.
 * Note: It is OK to sleep while holding a beta semaphore.
 */
...

/*
 * Release the beta semaphore when done.
 */
b_vsema(&mydrv_sema);
```

SEE ALSO

b_cpsema(KER2), b_initsema(KER2), b_owns_sema(KER2),
b_psema(KER2),
NAME

cmp(KER2) – Compare two byte arrays

SYNOPSIS

#include <sys/kern_svcs.h>

int cmp (void * s1, void * s2, size_t n);

PARAMETERS

s1 Pointer to the first byte array.

s2 Pointer to the second byte array.

n Number of bytes to compare.

DESCRIPTION

The cmp() kernel function compares n bytes of the byte arrays starting at s1 and s2. If these n bytes are identical, return zero. If the n bytes are not identical, the integer value of (s1[k]−s2[k]) is returned, where k is the failing byte offset in the array. Unlike strcmp(), cmp() does not terminate when it encounters a null byte.

RETURN VALUES

0 The byte arrays are identical.

<>0 The byte arrays are different.

CONSTRAINTS

SEE ALSO

strcmp(KER2)
NAME

bcopy(KER2) – Copy data from a source buffer to a destination buffer

SYNOPSIS

#include <sys/kern_svcs.h>

void bcopy (void *from, void *to, size_t n);

PARAMETERS

from Pointer to the source buffer.
to Pointer to the destination buffer.
n Number of bytes to copy.

DESCRIPTION

The bcopy() kernel function copies n bytes from a kernel space buffer to another kernel space buffer. The two buffers must not overlap.

To copy data between user space and kernel space, use copyin(), copyout(), or uiomove().

To copy data between buffers in other user spaces, use privlbcopy().

RETURN VALUES

None

CONSTRAINTS
WARNINGS

The function `bcopy()` must not be used for transfers between memory and IO space. The underlying routines make choices of the optimal transfer code which will probably not be supported on either the IO bus adapter or the attached interface card. Failure to heed this warning may result in data corruption, an HPMC (High Priority Machine Check), or a call to `panic()`.

SEE ALSO

`copyin(KER2)`, `copyout(KER2)`, `privlbcopy(KER2)`, `uiomove(KER2)`
**NAME**

`biodone(KER2)` – Complete the buffer I/O transaction.

**SYNOPSIS**

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

void biodone (struct buf * bp);
```

**PARAMETERS**

- `bp` A pointer to a `buf` structure.

**DESCRIPTION**

The `biodone()` kernel function completes the buffer I/O transaction. There should be a corresponding call to `biowait()` for the same `bp`.

If `B_CALL` is set in `bp->b_flags`, then `biodone()` calls the callback function specified in `bp->b_iiodone`. The callback function is expected to set the `B_DONE` flag in `bp->b_flags`.

If `B_CALL` is not set in `bp->b_flags`, then `biodone()` marks the buffer I/O as completed by setting the `B_DONE` flag in `bp->b_flags`. If `B_ASYNC` is set, then `biodone()` releases the `buf` structure and associated buffer pointed to by `bp`, else it resumes the thread waiting on the corresponding call to `biowait()`

**RETURN VALUES**

None.

**CONSTRAINTS**

Must not be called while holding a spinlock of order $\geq$ `BUF_HASH_LOCK_ORDER`.

**WARNINGS**

`biodone()` calls `panic()` if `B_DONE` is set in `bp->b_flags` upon entry.
EXAMPLE

/*
 * As a sanity check, make sure that B_DONE is not set
 * in b_flags before we call biodone().  If B_DONE is
 * set, then we must be holding a stale buf structure.
 */
VASSERT(!(bp->b_flags & B_DONE));

/*
 * Return I/O completion info in the buf structure.
 */
if (transfer_error) {
    bp->b_error = EIO;
    bp->b_flags |= B_ERROR;
} else {
    bp->b_resid = transfer_residue;
}

/*
 * Complete the buffer I/O transaction. Typically, this
 * results in awaking the thread sleeping in biowait().
 */
biodone(bp);

SEE ALSO

biowait(KER2), buf(KER4)
NAME

biowait(KER2) – Wait for the buffer I/O to complete.

SYNOPSIS

#include <sys/buf.h>

int biowait (struct buf * bp);

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

The biowait() kernel function waits for the completion of the buffer I/O specified by bp. A corresponding call to biodone() is required to resume the waiting thread.

RETURN VALUES

0 Successful completion.
<>0 Error.

CONSTRAINTS

Must not be called in an interrupt context.
Must not be called while holding a spinlock.
EXAMPLES

```c
int error;
struct buf *bp;
...

/*
 * After starting the I/O request, wait for its completion.
 */
error = biowait(bp);

/*
 * biowait() returns 0 if the I/O completes successfully.
 * A non-zero value is returned if an error has been
 * encountered, however, the error value returned is not
 * always for the I/O completion. To get the I/O
 * completion error that is returned with the buf, we
 * need to call geterror().
 */
if (error) {
    error = geterror(bp);
}
```

SEE ALSO

biodone(KER2), buf(KER4), geterror(KER2)
NAME

brelse(KER2) – Release a buffer to the buffer cache.

SYNOPSIS

#include <sys/buf.h>

void brelse (struct buf * bp);

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

The brelse() kernel function releases a buffer to the buffer cache. The
buffer header (buf structure) is pointed to by bp and it should have been
previously allocated by a call to geteblk(). If there are threads waiting
for this or any free buffer in the buffer cache, the waiting threads are
awakened by brelse().

The B_BUSY and B_BCACHE flags must be set in bp->b_flags upon entry
to brelse().

RETURN VALUES

None.

CONSTRAINTS

Must not be called while holding a spinlock of order >=
BUF_FREE_LOCK_ORDER

SEE ALSO

buf(KER4), geteblk(KER2)
NAME

buf(KER4) – File system buffer header structure for block I/O

SYNOPSIS

#include <sys/buf.h>

DESCRIPTION

The buf structure describes a file system buffer header used for block I/O. The buffer header contains a pointer to the system buffer allocated to the header and specifies control and status information for the I/O transfer to be performed. Block drivers are passed buffer headers through their driver_strategy routines. Buffer headers and their associated system buffers may be allocated by drivers through geteblk().

The buf structure also describes a buffer header used for raw I/O. The buffer header in this case is formatted by physio() and points to a user buffer, not a system buffer. The user buffer is mapped into kernel space for legacy drivers that specify C_MAP_BUFFER_TO_KERNEL in the d_flags field of their drv_opts_t structure.

Do not depend on the size of the buf structure when writing a driver. In particular, do not allocate a buf structure through kmalloc() and related memory allocation functions. Doing so may invalidate binary compatibility with future releases of HP-UX.

STRUCTURE MEMBERS

The buf structure is defined in <sys/buf.h>. The following table lists important fields in the buffer and their types.

Table 2-1 Relevant buf Structure Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct buf *</td>
<td>av_back</td>
</tr>
<tr>
<td>struct buf *</td>
<td>av_forw</td>
</tr>
<tr>
<td>int32_t</td>
<td>b_blkno</td>
</tr>
</tbody>
</table>
Table 2-1 Relevant buf Structure Fields (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>daddr_t</td>
<td>b_b_blkno</td>
</tr>
<tr>
<td>int32_t</td>
<td>b_bufsize</td>
</tr>
<tr>
<td>dev_t</td>
<td>b_dev</td>
</tr>
<tr>
<td>short</td>
<td>b_error</td>
</tr>
<tr>
<td>int32_t</td>
<td>b_flags</td>
</tr>
<tr>
<td>int (*)()</td>
<td>b_iiodone</td>
</tr>
<tr>
<td>struct buf *</td>
<td>b_merge</td>
</tr>
<tr>
<td>uint16_t</td>
<td>b_merge_cnt</td>
</tr>
<tr>
<td>unsigned int</td>
<td>b_resid</td>
</tr>
<tr>
<td>intptr_t</td>
<td>b_s2</td>
</tr>
<tr>
<td>char</td>
<td>b_s3</td>
</tr>
<tr>
<td>intptr_t</td>
<td>b_s7</td>
</tr>
<tr>
<td>intptr_t</td>
<td>b_s8</td>
</tr>
<tr>
<td>space_t</td>
<td>b_spaddr</td>
</tr>
<tr>
<td>caddr_t</td>
<td>b_un.b_addr</td>
</tr>
<tr>
<td>uint16_t</td>
<td>b2_flags</td>
</tr>
</tbody>
</table>

**av_back av_forw**

Backward and forward pointers in the buffer headers on the free list where the B_BUSY flag is not sent in b_flags. Drivers can use the av_forw and av_back pointers to maintain a queue of busy buffer headers.

**b_bcount**

Number of bytes to be transferred.

**b_blkno**

Block number of the first logical block to be accessed on the target device. A block contains DEV_BSIZE bytes.

**b_bufsize**

Size of the allocated buffer.
b_dev
The dev_t major and minor numbers of the target device.

b_error
If the B_ERROR bit is set in b_flags, b_error contains the errno value for the error that occurred. This field is set by the driver before calling biodone() and is obtained by calling geterror().

b_flags
Information about the buffer. This value is composed from the following bit flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>If set…</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_ASYNC</td>
<td>Buffer write is synchronous. Do not wait for I/O completion. Mutually exclusive with B_SYNC.</td>
</tr>
<tr>
<td>B_BCACHE</td>
<td>The buffer is allocated from the file system buffer cache.</td>
</tr>
<tr>
<td>B_BUSY</td>
<td>The buffer is in use.</td>
</tr>
<tr>
<td>B_CACHE</td>
<td>bread() located this buffer in the cache.</td>
</tr>
<tr>
<td>B_CALL</td>
<td>iodone() is to call the function pointed to by b_iodone.</td>
</tr>
<tr>
<td>B_DELWRI</td>
<td>Delayed write. Write at exit of avail list processing by the buffer cache management code.</td>
</tr>
<tr>
<td>B_DONE</td>
<td>The buffer transfer has completed. biodone() sets this flag.</td>
</tr>
<tr>
<td>B_END_OF_DATA</td>
<td>This flag is used to terminate, without error, a physio transfer, with less than b_count bytes transferred.</td>
</tr>
<tr>
<td>B_ERROR</td>
<td>An error occurred during the I/O transfer. If the driver sets this flag, it must also set the b_error field with an errno value.</td>
</tr>
<tr>
<td>B_FSYSIO</td>
<td>Buffer came from bread() or bwrite().</td>
</tr>
<tr>
<td>B_INVAL</td>
<td>The buffer does not contain valid information.</td>
</tr>
</tbody>
</table>
**buf(KER4)**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>B_NDELAY</strong></td>
<td>Do not retry on failures.</td>
</tr>
<tr>
<td><strong>B_NOCACHE</strong></td>
<td>Do not cache data buffer when released.</td>
</tr>
<tr>
<td><strong>B_PAGEOUT</strong></td>
<td>This flag is used by the buffer cache management system and should not be touched by a driver.</td>
</tr>
<tr>
<td><strong>B_PFTIMEOUT</strong></td>
<td>With this flag set, a driver is expected to return the I/O request with b_error set to EPOWERF if the device has experienced a power failure. Drivers typically employ a timeout mechanism to detect a device power failure during an I/O transfer.</td>
</tr>
<tr>
<td><strong>B_PHYS</strong></td>
<td>Indicates the buffer is a user buffer. This flag is normally set by physio().</td>
</tr>
<tr>
<td><strong>B_PRIVATE</strong></td>
<td>Indicates the buffer header is private to a subsystem such as LVM.</td>
</tr>
<tr>
<td><strong>B_RAW</strong></td>
<td>Indicates the buffer header is sent to a character (raw) device.</td>
</tr>
<tr>
<td><strong>B_READ</strong></td>
<td>Data are to be read from the device to host memory. If B_READ is not set, data are to be written from host memory to the device.</td>
</tr>
<tr>
<td><strong>B_REWRITE</strong></td>
<td>This flag is used by the buffer cache management system and should not be touched by a driver.</td>
</tr>
<tr>
<td><strong>B_ASYNC</strong></td>
<td>Buffer write is synchronous. Wait for I/O completion. Mutually exclusive with B_ASYNC.</td>
</tr>
<tr>
<td><strong>B_WANTED</strong></td>
<td>One or more threads are sleeping on the buffer header, waiting for the buffer to be freed.</td>
</tr>
</tbody>
</table>
B_WRITE  A pseudo flag that semantically indicates 'not B_READ'. The value of B_WRITE is 0; it has no testable bits. To test for a write request, test for the absence of B_READ:

```c
if (!(bp->b_flags & B_READ) )
```

The expression `(bp->b_flags & B_WRITE)` is always zero.

B_WRITEV  This flag is used by LVM when attempting to correct disk soft errors and should not be touched by a driver.

b_iiodone  Pointer to a function that iiodone() calls to complete the I/O request if the B_CALL flag is set in b_flags. The function takes a pointer to the buffer header as its argument and is expected to set the B_DONE flag in b_flags.

b_merge  Pointer to the next buf structure where the list of buffers have been logically merged together. Valid only when B2_LIST is set in b2_flags.

b_merge_cnt  Number of buffers merged together through b_merge.

b_resid  Number of bytes remaining to be transferred; usually set to zero after a successful transfer. The driver_strategy routine sets this field before calling biodone().

b_s2  Scratch field for driver use. For example, SCSI Interface Drivers use this field to store a pointer to a SCSI Control Block.

b_s3  Scratch field for driver use. For example, the SCSI Subsystem uses this field to store state information.

b_s7  Scratch field for driver use. For example, the SCSI Subsystem reserves this field for device drivers.

b_s8  Scratch field for driver use. For example, the SCSI subsystem reserves this field for device drivers.

b_spaddr  Space ID of the buffer specified by the buffer header. Do not assume this value to be KERNELSPACE.
b_un.b_addr Virtual address of the buffer specified by the buffer header. The buffer may be mapped in kernel space or it may be mapped in user space. If mapped in user space, b_spaddr must be used with b_addr to form the global virtual address in order to access the buffer - see privlbcopy().

b2_flags Information about the buffer in addition to b_flags. This value is composed from the following bit flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>If set...</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2_LIST</td>
<td>Buffer is linked with other buffers through the b_merge field.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

biodone(KER2), biowait(KER2), brelse(KER2), geteblk(KER2), geterror(KER2), physio(KER2), privlbcopy(KER2)
NAME

busywait(KER2) – Wait at least $t$ microseconds by spinning processor cycles

SYNOPSIS

```c
#include <sys/kern_svcs.h>
#include <sysio.h>

void busywait (ulong_t $t$);
```

PARAMETERS

$t$ The wait time in microseconds.

DESCRIPTION

The `busywait()` kernel function waits at least $t$ microseconds by spinning processor cycles. The processor does no useful work while busy waiting, so $t$ should be kept as short as possible.

`busywait()` does not block (i.e., sleep) and can be called in an interrupt context or while holding a spinlock. If `busywait()` is entered with external interrupts enabled, it is possible for an interrupt to occur and make the actual wait time far exceed the specified $t$ microseconds.

RETURN VALUES

None.

CONSTRAINTS

EXAMPLES

SEE ALSO
NAME

bzero(KER2) – Fill a kernel buffer with zeros.

SYNOPSIS

#include <sys/kern_svcs.h>

void bzero (void * addr, size_t n);

PARAMETERS

addr Address of kernel buffer.

n Number of bytes to be zeroed.

DESCRIPTION

The bzero() kernel function writes $n$ contiguous bytes of zero, starting at the kernel address specified by $addr$.

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

bcopy (KER2)
NAME

`copyin` (KER2) – Copy data from a user buffer to a kernel buffer

SYNOPSIS

```c
#include <sys/kern_svcs.h>

int copyin(void *from_user, void *to_kernel, size_t n);
```

PARAMETERS

- `from_user`: Source user space address.
- `to_kernel`: Destination kernel space address.
- `n`: Number of bytes to copy.

DESCRIPTION

The `copyin()` kernel function copies `n` bytes of data from the user space address `from_user` to the kernel space address `to_kernel`. The call to `copyin()` must be made while executing in the user context; that is, while executing in the top half of the driver where the user invokes the driver via a system call such as `ioctl()`.

`copyin()` may block (i.e., sleep), so it must not be called while holding a spinlock.

RETURN VALUES

- `0`: Successful completion.
- `<>0`: Error.

CONSTRAINTS

- Must be called in an interrupt context.
- Must be called while holding a spinlock.
WARNINGS

The kernel stack is limited in size. If the buffer to be copied is larger than 128 bytes, avoid allocating buffer space on the kernel stack; instead, allocate the buffer from kernel memory (e.g., by calling kmalloc()). When large buffers are allocated on the kernel stack, the kernel stack may overflow and cause the kernel to panic.

EXAMPLE

```c
char my_buff[128];
/*
 * Copy from the user buffer to my_buff[] on the kernel stack.
 * Note that buffers larger than 128 bytes should be
 * allocated from kernel memory by calling kmalloc().
 */
if (copyin(user_buf, my_buf, sizeof(my_buf))) {
    return EFAULT;
}
```

SEE ALSO

bcopy(KER2), copyout(KER2), privlbcopy(KER2), uiomove(KER2)
NAME

\texttt{copyout}(KER2) – Copy data from a kernel buffer to a user buffer

SYNOPSIS

\begin{verbatim}
#include <sys/kern_svcs.h>

int copyout (void *from_kernel, void *to_user, size_t n);
\end{verbatim}

PARAMETERS

\begin{itemize}
  \item \texttt{from_kernel} \hspace{1cm} Source kernel space address.
  \item \texttt{to_user} \hspace{1cm} Destination user space address.
  \item \texttt{n} \hspace{1cm} Number of bytes to copy.
\end{itemize}

DESCRIPTION

The \texttt{copyout()} kernel function copies \textit{n} bytes of data from the kernel space address \texttt{from_kernel} to the user space address \texttt{to_user}. The call to \texttt{copyout()} must be made while executing in the user context; that is, while executing in the top half of the driver where the user invokes the driver via a system call such as \texttt{ioctl()}. \texttt{copyout()} may block (i.e., sleep), so it must not be called while holding a spinlock.

RETURN VALUES

\begin{itemize}
  \item 0 \hspace{1cm} Successful completion.
  \item \textless{}0 \hspace{1cm} Error.
\end{itemize}

CONSTRAINTS

Must be called in an interrupt context.
Must not be called while holding a spinlock.
WARNINGS

The kernel stack is limited in size. If the buffer to be copied is larger than 128 bytes, avoid allocating buffer space on the kernel stack; instead, allocate the buffer from kernel memory (e.g., by calling kmalloc()). When large buffers are allocated on the kernel stack, the kernel stack may overflow and cause the kernel to panic.

EXAMPLE

```c
char my_buff[128];

/*
 * Copy to the user buffer from my_buff[] on the kernel stack.
 * Note that buffers larger than 128 bytes should be
 * allocated from kernel memory by calling kmalloc().
 */
if (copyout(my_buf, user_buf, sizeof(my_buf))) {
    return EFAULT;
}
```

SEE ALSO

bcopy(KER2), copyin(KER2), privlbcopy(KER2), uiomove(KER2)
NAME

cspinlock(KER2) – Conditionally acquire (lock) a spinlock.

SYNOPSIS

#include <sys/spinlock.h>

int cspinlock (lock_t * lock);

PARAMETERS

lock Pointer to a lock_t structure.

DESCRIPTION

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread’s current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The cspinlock() kernel function attempts to conditionally acquire (lock) a spinlock pointed to by lock. The calling thread does not spin if the spinlock is currently owned.

RETURN VALUES

0 The lock is already owned by this processor or another processor.
1 The lock has been acquired by this processor.

CONSTRAINTS
EXAMPLE

while (list_entry != list_head) {
    /* Get the next list entry protected by mydrv_lock_2. */
    spinlock(mydrv_lock_2);
    list_entry = list_head->next_entry;
    if (list_entry == list_head) {
        spinunlock(mydrv_lock_2);
        break; /* at end of list */
    }

    /* We need to acquire another spinlock, but we are
    * acquiring the two locks in reverse order. To avoid
    * a deadlock, we conditionally attempt to acquire
    * the next spinlock with cspinlock(). If already
    * owned, we must release the other spinlock.
    * Note: cspinlock() does not check the lock order.
    */
    if (!cspinlock(mydrv_lock_1)) {
        /* Failed to acquire the next spinlock.
         * As a sanity check, assert that we are not
         * on the processor that owns the spinlock.
         */
        VASSERT(!owns_spinlock(mydrv_lock_1));
        /* Release mydrv_lock_2 to allow the processor
         * holding mydrv_lock_1 to make forward progress.
         */
        spinunlock(mydrv_lock_2);
        /* Get the two locks in the correct lock order.
         */
        spinlock(mydrv_lock_1);
        spinlock(mydrv_lock_2);
    }

    /* We now have both locks. Check to be sure the
    * list_head did not change if we had to release
    * mydrv_lock_2 to avoid a deadlock.
    */
if (list_entry != list_head->list_entry) {
    /*
     * Release the two locks and try again.
     */
    spinunlock(mydrv_lock_2);
    spinunlock(mydrv_lock_1);
    continue;
}

/**
 * Process the list entry.
 */
...

/*
 * Release the two locks and get the next entry.
 */
    spinunlock(mydrv_lock_2);
    spinunlock(mydrv_lock_1);
} /* end of while */

SEE ALSO

alloc_spinlock(KER2), dealloc_spinlock(KER2), owns_spinlock(KER2), spinlock(KER2), spinunlock(KER2)
NAME
dealloc_spinlock(KER2) – Deallocate a spinlock resource.

SYNOPSIS

#include <sys/spinlock.h>

void dealloc_spinlock (lock_t * lock);

PARAMETERS

lock Pointer to a lock_t structure.

DESCRIPTION

Spinlocks are the basic locking primitives used by the kernel for short-term locks. When a thread acquires a spinlock, the thread’s current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The dealloc_spinlock() kernel function deallocates a spinlock resource, pointed to by lock, which was previously allocated by alloc_spinlock().

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

alloc_spinlock(KER2), cspinlock(KER2), get_sleep_lock(KER2), owns_spinlock(KER2), spinlock(KER2), spinunlock(KER2)
NAME

FREE(KER2) – Kernel macro to deallocate kernel memory.

SYNOPSIS

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

#define FREE (addr, type)
```

PARAMETERS

- `addr` Address of the kernel memory previously allocated by `MALLOC()`.
- `type` The memory allocation pool type.

DESCRIPTION

The `MALLOC()` and `FREE()` kernel macros are deprecated interfaces and may be obsoleted in a future release of HP-UX. Use the kernel functions `kmalloc()` and `kfree()` in place of `MALLOC()` and `FREE()`.

The `FREE()` kernel macro deallocates (frees) kernel memory previously allocated by `MALLOC()`. The `addr` parameter must be the address returned by `MALLOC()` and `type` must be the same type passed to `MALLOC()`.

CONSTRAINTS
EXAMPLES

struct buf * bp;

/*
 * Allocate a buf structure for internal use.
 */
MALLOC(bp, struct buf *, sizeof(struct buf),
       M_IOSYS, M_NOWAIT);
...

/*
 * When done with the buf structure, release it.
 */
FREE(bp, M_IOSYS);

SEE ALSO

kmallo(KER2), kfree(KER2), MALLOC(KER2)
NAME

get_sleep_lock(KER2) – Acquire a sleep queue spinlock.

SYNOPSIS

#include <sys/spinlock.h>

lock_t * get_sleep_lock (void * chan);

PARAMETERS

chan Channel parameter passed to sleep() and wakeup().

DESCRIPTION

The get_sleep_lock() kernel function acquires the spinlock that protects the sleep queue associated with the channel chan. Different values of chan may map onto different sleep queues. A thread calls get_sleep_lock() before it calls sleep(). The sleep queue spinlock associated with chan is released by the corresponding call to sleep().

get_sleep_lock() synchronizes the race condition between sleep() and wakeup(). It is possible for wakeup() to be called on another processor before a thread can be put to sleep. However, the corresponding wakeup() must acquire the sleep queue spinlock held by the thread being put to sleep, thus it spins and waits. When wakeup() does acquire the spinlock, the thread will be on the sleep queue and ready to be awakened.

get_sleep_lock() may optionally be used as a synchronization wrapper around wakeup(). For example, consider the following code:

    (void) get_sleep_lock(wait_chan);
    start_async_activity();
    activity_count++;
    (void) sleep(wait_chan, PRIBIO);

Notice that activity_count is protected by a sleep queue spinlock which is released by sleep(). When the async activity completes, the corresponding completion routine calls get_sleep_lock() before it decrements activity_count and calls wakeup().
sleep_lock = get_sleep_lock(wait_chan);
if (activity_count) {
    activity_count--;
    (void)wakeup(wait_chan);
}
spinunlock(sleep_lock);

wakeup() can handle the case where the sleep queue spinlock is held by
the caller upon entry. The sleep queue spinlock must be explicitly
released by a call to spinunlock() when used in the above manner.

RETURN VALUES

get_sleep_lock() returns a pointer to the lock_t structure which
contains the spinlock that protects the sleep queue.

CONSTRAINTS

Must not be called while holding a spinlock of order >=
SLEEP_Q_LOCK_ORDER.

EXAMPLES

/*
 * The sleeping thread waits on a channel by first calling
 * get_sleep_lock(), then it starts an asynchronous activity
 * and finally calls sleep() which releases the sleep lock.
 */
(void)get_sleep_lock(wait_chan);
start_async_activity();
(void)sleep(wait_chan, PRIBIO);
...

/*
 * When the asynchronous activity completes (typically in an
 * interrupt service routine), the sleeping thread is awakened.
 */
(void)wakeup(wait_chan);

SEE ALSO

sleep(KER2), wakeup(KER2)
NAME

getc(KER2) – Manipulate characters on a clist

SYNOPSIS

#include <sys/clist.h>

int getc (struct clist * list);

PARAMETERS

list Pointer to a clist.

DESCRIPTION

The getc() kernel function returns the next character in the clist list.

RETURN VALUES

c The next character in the clist.
-1 If no characters are in the clist.

CONSTRAINTS

EXAMPLES

void
catq(struct clist * from,
     struct clist * to)

/* concatenate characters from one clist to another */

{
  int c;

  while ((c = getc(from)) >= 0)
    (void) putc(c, to);
}
Kernel Reference Pages
getc(KER2)

SEE ALSO

putc(KER2), getcb(KER2), getcf(KER2), putcb(KER2), putcf(KER2)
NAME

getcb(KER2) – Manipulate cblocks on a clist

SYNOPSIS

#include <sys/clist.h>

struct cblock * getcb (struct clist * list);

PARAMETERS

list Pointer to a clist.

DESCRIPTION

getcb() removes the last cblock from the clist pointed to by list.

RETURN VALUES

<>NULL Successful completion. The value is a pointer to the cblock that was removed.

NULL Error.

CONSTRAINTS

EXAMPLE

int
mydev_open(dev_t dev, int flag)
{
    struct cblock *cp;
    struct mydevice dv;

    /* Flush out the device buffer */
    while ((cp = getcb(&dv->queue)) != NULL)
        putcf(cp);
    ...
}
Kernel Reference Pages
getcb(KER2)

SEE ALSO

putcb(KER2), getc(KER2), getcf(KER2), putc(KER2), putcf(KER2)
NAME

getcf(KER2) – Get a cblock on the cblock free list

SYNOPSIS

#include <sys/clist.h>

struct cblock * getcf (void);

PARAMETERS

None

DESCRIPTION

getcf() gets the next free cblock on the cfreelist.

RETURN VALUES

<>NULL Successful completion. The value is a pointer to the cblock.

NULL Error.

CONSTRAINTS

SEE ALSO

getc(KER2), getcb(KER2)
NAME

geteblk(KER2) – Allocate a buffer from the buffer cache.

SYNOPSIS

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

struct buf * geteblk (int size);
```

PARAMETERS

```c
size
```
The size in bytes of the requested block.

DESCRIPTION

The `geteblk()` kernel function allocates a buffer large enough to contain `size` bytes of data from the buffer cache and returns a pointer to the buffer header. The `size` parameter must be <= MAXBSIZE defined in `<sys/param.h>`.

Fields in the `buf` structure returned are initialized as follows:

- `b_flags` is set with `B_BUSY`, `B_BCACHE` and `B_INVAL` flags
- `b_un.b_addr` contains the virtual address of the allocated buffer
- `b_spaddr` contains the space address (space ID) of the allocated buffer

Drivers do not normally allocate buffers. However, to implement special features, such as `ioctl()` commands, the driver may need its own buffer space. Buffers allocated by calling `geteblk()` should be used for short durations and released by calling `brelse()` when no longer needed.

RETURN VALUES

`geteblk()` returns a pointer to the allocated buffer header.

CONSTRAINTS

- Must not be called in an interrupt context.
- Must not be called while holding a spinlock.
EXAMPLE

```c
struct buf * bp;

/*
 * Allocate a buffer that can store a page of data.
 */
bp = geteblk(PAGESIZE);
VASSERT(bp);
```

SEE ALSO

brelse(KER2), buf(KER4)
NAME
geterror(KER2) – Return the error number from the buffer header.

SYNOPSIS

#include <sys/buf.h>

int geterror (struct buf * bp);

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

geterror() returns the error number from the b_error field of the
buffer header structure pointed to by bp. If B_ERROR is not set in
bp->b_flags, the I/O is assumed to have completed successfully.

RETURN VALUES

0 I/O completed successfully.
<>0 Error number.

CONSTRAINTS
EXAMPLE

```c
int error;
struct buf *bp;
...

/*
 * After starting the I/O request, wait for its completion.
 */
error = biowait(bp);

/*
 * biowait() returns 0 if the IO completes successfully.
 * A non-zero value is returned if an error has been
 * encountered, however, the error value returned is not
 * always for the IO completion. To get the IO
 * completion error that is returned with the buf, we
 * need to call geterror().
 */
if (error) {
    error = geterror(bp);
}
```

SEE ALSO

biowait(KER2), buf(KER4)
NAME

gsignal(KER2) – Send the specified signal to all processes in a process group

SYNOPSIS

#include <h/proc_iface.h>

void gsignal (pid_t pgrp, int sig);

PARAMETERS

pgrp Process group identifier.
sig Signal number.

DESCRIPTION

The gsignal() kernel function sends the signal sig to all processes that have pgrp as their process group identifier.

The pgrp parameter is returned by the kernel function p_grp().

Signal numbers are defined in <sys/signal.h>

RETURN VALUES

None.

CONSTRAINTS

EXAMPLES

cinclude <sys/user.h>
#include <sys/signal.h>
#include <h/proc_iface.h>

/*
 * Signal all processes in the current process group.
 */
gsignal(p_pgrp(u.u_procp), SIGIO);
SEE ALSO

p_pgrp(KER2), psignal(KER2)
NAME

iovec(KER4) – Data buffer descriptor for character I/O and WSIO CDIO mapping services.

SYNOPSIS

#include <sys/uio.h>

DESCRIPTION

The iovec kernel structure points to the data buffer for character I/O. It can be declared and used to map a kernel data buffer for an I/O bus master and it is a member of the uio structure used by uiomove().

STRUCTURE MEMBERS

The iovec structure is defined in <sys/uio.h>. It has the following fields. Their data types are shown in the table below. Drivers must not modify any fields of this structure when using uiomove(). When this structure is used for mapping with a WSIO CDIO mapping service, the driver initially sets up iov_base and iov_len.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>caddr_t</td>
<td>iov_base</td>
</tr>
<tr>
<td>size_t</td>
<td>iov_len</td>
</tr>
</tbody>
</table>

iov_base The address of the user's buffer for uiomove or kernel buffer for WSIO CDIO mapping services.

iov_len The number of bytes to be transferred.

SEE ALSO

uio(KER4), wsio_fastmap(WSIO3), wsio_map(WSIO3), wsio_remap(WSIO3)
NAME

`kernel_iomap(KER2)` — Map physical space to host virtual space with user read/write protection.

SYNOPSIS

```c
#include <sys/iomap.h>

caddr_t kernel_iomap (caddr_t virt_addr, caddr_t phys_addr, pgcnt_t count, prot_t prot);
```

PARAMETERS

- `virt_addr` NULL or equivalent to `phys_addr`
- `phys_addr` The start of the host physical address (not the bus relative address) for an area of memory on an I/O bus (EISA, for example).
- `count` The length of the mapping in pages (4 Kbyte pages).
- `prot` The protection to be used for the mapping which should be `PROT_URW`.

DESCRIPTION

The kernel routine `kernel_iomap()` is used to map physical space onto a host virtual address for cards with memory components that need to be mapped in for user read/write access. It maps the space with user read/write protection.

RETURN VALUES

- `<NULL` Successful completion. The value is the host virtual address for accessing the space specified by the parameters.
- `NULL` Error.
CONSTRAINTS

EXAMPLES

/*
** example maps 1 page of I/O space at 0xf0000000 with
** user read/write protections (PROT_URW)
*/

#include <sys/iomap.h>
#include <sys/mman.h>
#include <machine/psl.h>

{
    int sm;
    char byte;
    caddr_t mapped_addr;
    caddr_t phys_addr = 0xf0000000;
    u_int count = 1;

    /* map the space to the driver */
    mapped_addr =
        kernel_iomap(NULL, phys_addr, count, PROT_URW);
    if (mapped_addr == NULL)
        goto map_failed; /* bailout code for error */
    /* grant kernel access to user mapped space */
    sm = rsm(PSW_P);
    byte = *mapped_addr; /* read first byte of space */
    /* other access to user mapped I/O space */
    /* deny kernel access to space */
    if (sm & PSW_P)
        ssm(PSW_P);
    /*
    ** Note - even though the space is mapped with user
    ** read/write protections, this does NOT mean that a
    ** user process can access the space. Your driver
    ** should call user_iomap, to map the space to a
    ** given process.
    */
}


WARNINGS

The use of `kernel_iomap()`, which uses `PROT_URW`, effectively disables kernel mode access of the mapped space. If the driver requires access to the mapped space then the `rsm(PSW_P)` and `ssm(PSW_P)` instructions must be used because the I/O pages are mapped with a non-zero protection ID.

In the case of EISA (not ISA) cards, the EISA IO space is mapped (prior to calling the driver attach routine) with `PROT_KRW`. This mapping is for one page (4 Kbytes). If you desire user access to EISA IO space you will need to first unmap `isc->if_reg_ptr` with a call to `unmap_mem_from_host()`, and then remap the space with `kernel_iomap_public()`. A similar situation exists with the PCI bus memory space.

Hardware disables caching for all EISA data accesses, including EISA memory space.

`kernel_iomap()` will return `NULL` if a request overlaps an existing mapping. If the request exactly overlays (same start address and size) an existing mapping, the mapping services will return the host virtual address; n.b., second and subsequent mappings inherit the protection values of the first mapping call.

SEE ALSO

`kernel_iomap_public(KER2)`, `kernel_iounmap(KER2)`, `user_iomap(KER2)`, `user_iounmap(KER2)`
NAME

\texttt{kernel_iomap_public(KER2)} – Map physical bus to host virtual space with no protection.

SYNOPSIS

\begin{verbatim}
#include <sys/iomap.h>

caddr_t kernel_iomap_public (caddr_t virt_addr, caddr_t phys_addr,
                         pgcnt count, prot_t prot);
\end{verbatim}

PARAMETERS

\begin{tabular}{ll}
\textit{virt\_addr} & NULL or equivalent to \textit{phys\_addr} \\
\textit{phys\_addr} & The start of the host physical address (not the bus relative address) for an area of memory on an I/O bus (EISA, for example). \\
\textit{count} & The size of the memory space in pages (4 Kbyte pages). \\
\textit{prot} & The protection to be used for the mapping. Should be \texttt{PROT\_URW}. 
\end{tabular}

DESCRIPTION

The kernel routine \texttt{kernel_iomap_public()} is used to map physical space onto a host virtual address for cards with memory components that need to be mapped in for user read/write access. When called with \textit{prot} = \texttt{PROT\_URW}, it maps the space with no protection bits set. The behavior of this call is very similar to \texttt{kernel_iomap()} with the exception that drivers can access the mapped pages without using the \texttt{rsm}(PSW\_P) and \texttt{ssm}(PSW\_P) instructions.

RETURN VALUES

\begin{tabular}{ll}
\texttt{<>NULL} & Successful completion. The value is the host virtual address for accessing the space specified by the parameters. \\
\texttt{NULL} & Error. 
\end{tabular}
CONSTRANTS

EXAMPLES

/*
** example maps 1 page of I/O space at
** 0xf0000000 with no protection
*/
#include <sys/iomap.h>
#include <sys/mman.h>
.
{
    int sm;
    char byte;
    caddr_t mapped_addr;
    caddr_t phys_addr = 0xf0000000;
    u_int count = 1;

    /* map the space to the driver */
    mapped_addr =
        kernel_iomap_public(NULL,phys_addr,count,PROT_URW);
    if (mapped_addr == NULL)
        goto map_failed; /* bailout code for error */
    byte = *mapped_addr; /* read first byte of space */
    /* other access to user mapped I/O space */
    /*
    ** Note - even though the space is mapped with user
    ** read/write proections, this does NOT mean that a
    ** user process can access the space. Your driver
    ** should call user_iomap, to map the space to a
    ** given process. Alternatively, the user process
    ** can call the iomap driver
    */
    
}
WARNINGS

When called with PROT_URW protection mode (prot parameter), this call creates an unprotected mapping which allows I/O access from unauthorized processes.

This call should be used with extreme caution as an unauthorized process could read/write to any I/O space register mapped in this manner and crash/hang the system.

In the case of EISA (not ISA) cards, the EISA IO space is mapped (prior to calling the driver attach routine) with PROT_KRW. This mapping is for one page (4 Kbytes). If you desire user access to EISA IO space you will need to first unmap isc->if_reg_ptr with a call to unmap_mem_from_host(), and then remap the space with kernel_iomap_public(). A similar situation exists for PCI memory space.

Hardware disables caching for all EISA data accesses, including EISA memory space.

kernel_iomap_public() will return NULL if a request overlaps an existing mapping. If the request exactly overlays (same start address and size) an existing mapping, the mapping services will return the host virtual address requested.

SEE ALSO

kernel_iounmap(KER2), unmap_mem_from_host(KER2), user_iomap(KER2), user_iounmap(KER2)
NAME

kernel_iounmap(KER2) – Unmap physical bus from host virtual space

SYNOPSIS

#include <sys/iomap.h>

int kernel_iounmap (caddr_t virt_addr, pgcnt_t count);

PARAMETERS

virt_addr The virtual address previously obtained with a
kernel_iomap() or kernel_iomap_public() call

count The size of the memory space in pages (4 KByte pages).

DESCRIPTION

The kernel routine kernel_iounmap() is used to unmap previously
mapped virtual space.

RETURN VALUES

1 Error.

0 Success.

CONSTRAINTS

SEE ALSO

kernel_iomap(KER2), kernel_iomap_public(KER2),
user_iomap(KER2), user_iounmap(KER2)
NAME

kfree(KER2) – Deallocate kernel memory

SYNOPSIS

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

void kfree (void * addr, int type);
```

PARAMETERS

- **addr**: Address of the kernel memory previously allocated by a call to `kmalloc()`.
- **type**: The memory allocation pool type.

DESCRIPTION

The `kfree()` kernel function deallocates (frees) kernel memory previously allocated by a call to `kmalloc()`. The `addr` parameter must be the address returned by `kmalloc()` and `type` must be the same type passed to `kmalloc()`.

RETURN VALUES

None.

CONSTRAINTS

- None.

SEE ALSO

- `kmalloc(KER2)`
NAME
kmalloc(KER2) – Allocate kernel memory

SYNOPSIS

#include <sys/malloc.h>

void * kmalloc (size_t size, int type, arena_flags_t flags);

PARAMETERS

size The number of bytes (size) of kernel memory to allocate.
type The memory allocation pool type.
flags Flag to indicate the caller cannot block and wait for kernel memory availability.

DESCRIPTION

The kmalloc() kernel function allocates size bytes of kernel memory from the pool type specified. The memory allocated will always be contiguous in the virtual address space, but may be discontiguous in the physical address space. If size is greater than PAGESIZE, the physical pages allocated will probably not be physically contiguous.

Drivers should allocate memory from one of the following pool types: M_IOSYS or M_DMA. The M_IOSYS pool is for general I/O purposes; the M_DMA pool is for DMA purposes. Independent Hardware Vendors should specify the M_IHV pool for general I/O purposes of their drivers.

The flags parameter may optionally have the M_NOWAIT flag set. If M_NOWAIT is set and no memory is available from the requested pool type, the functions will return NULL. Without M_NOWAIT set, the caller can be blocked and made to wait for memory to become available.

The M_NOWAIT flag must be set if kmalloc() is:

- called in an interrupt context, or
- called while holding a spinlock.
If \texttt{M\_NOWAIT} is set, the caller must be prepared to handle the case where no kernel memory has been allocated.

\texttt{M\_WAITOK} should be passed as the \texttt{flags} parameter if the caller is able to block and wait for memory to become available.

**RETURN VALUES**

- \texttt{<>NULL} Virtual address of kernel memory allocated.
- \texttt{NULL} No kernel memory has been allocated. \texttt{NULL} is returned only when the \texttt{M\_NOWAIT} flag is set, otherwise the caller is blocked and waits until memory becomes available.

**CONSTRAINTS**

If the \texttt{M\_NOWAIT} flag is \textit{not} set:

- the caller must \textit{not} be in an interrupt context, and
- no spinlocks can be held

**SEE ALSO**

\texttt{kfree(KER2)}
NAME

Ktimeout(KER2) – Execute a callout function after a specified length of time at
driver level interrupt priority.

SYNOPSIS

```c
#include <sys/callout.h>
#include <sys/param.h>

callout_t * Ktimeout (int (*func)(), caddr_t arg, int t, void * dummy);
```

PARAMETERS

- `func` Function to execute when the time value `t` expires.
- `arg` Argument passed to the callout function `func`.
- `t` Time value in number of clock ticks.
- `dummy` Must be `NULL`.

DESCRIPTION

The Ktimeout() kernel function executes the specified callout function
`func` after `t` clock ticks have expired. Execution of `func` takes place in an
interrupt context at priority level 5 (driver level interrupt priority) where
external interrupts to the processor are disabled. Drivers are encouraged
to use the preferred interface timeout() where `func` is scheduled to
execute at priority level 2.

The `func` parameter is a pointer to a function that takes one argument. Although the prototype declares the function to return an int value, the
kernel does not make use of the return value.

The `arg` parameter is passed as the one argument to `func`.

The `t` parameter specifies the number of clock ticks to wait before calling
`func`. To express time in seconds, multiply `t` by `HZ`, where `HZ` is defined as
the number of clock ticks per second in `<sys/param.h>`.

The call to Ktimeout() returns immediately without waiting for the time
value `t` to expire. The timeout can be cancelled by making a
corresponding call to untimout().
RETURN VALUES

Ktimeout() returns a pointer to a callout structure.

CONSTRAINTS

Must not be called while holding a spinlock of order >= CALLOUT_LOCK_ORDER.

WARNINGS

Callout resources are not dynamically expandable. Each call to Ktimeout() allocates a callout resource, and the resource is not released until the time value expires or the timeout is cancelled. The kernel may panic if no callout resources are available.

EXAMPLES

/*
 * Set a timeout to call my_timeout_func() passing my_arg
 * after 5 seconds have expired.
 */
(void)Ktimeout(my_timeout_func, my_arg, 5*HZ, NULL);
...

static int
my_timeout_func(caddr_t arg)
{
    ...
}

SEE ALSO

timeout(KER2), untimeout(KER2)
**NAME**

`ldsid(KER2)` – Return (load) the space ID (SID) for a kernel virtual address.

**SYNOPSIS**

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

space_t ldsid (void * addr);
```

**PARAMETERS**

- `addr` Kernel virtual address

**DESCRIPTION**

The `ldsid()` kernel function returns the space ID (SID) for the kernel virtual address `addr`. The SID and virtual address are combined to form the global virtual address.

`ldsid()` may only be used with kernel virtual addresses. Do not call `ldsid()` for a user space address.

**RETURN VALUES**

`ldsid()` returns the space ID (SID) for a kernel virtual address.

**CONSTRAINTS**

**EXAMPLES**

```
/*
 * Copy data into the buffer specified through a buf (bp).
 * The b_spaddr and b_un.b_addr fields may contain the SID
 * and virtual address of a user space buffer. The from_addr
 * is a kernel virtual address and ldsid() returns its SID.
 */

privilbcopy(ldsid(from_addr), from_addr,
            bp->b_spaddr, bp->b_un.b_addr,
            sizeof(*from_addr));
```
Kernel Reference Pages

ldsid(KER2)

SEE ALSO

privlbcopy(KER2)
NAME

major(KER2) – Extract the major number from a device number

SYNOPSIS

#include <sys/sysmacros.h>

#define major(x) (((int)(((unsigned)(x)>>24)&0xff))

PARAMETERS

x

A dev_t device number.

DESCRIPTION

major() is a macro that returns the major number from the device number.

CONSTRAINTS

SEE ALSO

minor(KER2)
NAME
makedev(KER2) – Make a device number from major and minor numbers

SYNOPSIS

#include <sys/sysmacros.h>

#define makedev(x, y) ((dev_t)(((x)<<24) | (y & 0xffffff)))

PARAMETERS

x       A major number.
y       A minor number.

DESCRIPTION

The makedev() kernel macro builds a device number from major and minor numbers.

CONSTRAINTS

SEE ALSO

major(KER2), minor(KER2)
NAME

MALLOC(KER2) – Kernel macro to allocate kernel memory

SYNOPSIS

```c
#include <sys/malloc.h>

#define MALLOC(space, cast, size, type, flags)
```

PARAMETERS

- `space`: Pointer to the kernel memory allocated.
- `cast`: Type of the pointer `space` to be used as a cast in the macro.
- `size`: The number of bytes (size) of kernel memory to allocate.
- `type`: The memory allocation pool type.
- `flags`: Flag to indicate the caller cannot block and wait for kernel memory availability.

DESCRIPTION

The `MALLOC()` and `FREE()` kernel macros are deprecated interfaces and may be obsoleted in a future release of HP-UX. Use the kernel functions `kmalloc()` and `kfree()` in place of `MALLOC()` and `FREE()`.

The `MALLOC()` kernel macro allocates `size` bytes of kernel memory from the pool `type` specified. The memory allocated will always be contiguous in the virtual address `space`, but may be discontiguous in the physical address `space`. If `size` is greater than `PAGESIZE`, the physical pages allocated will probably not be physically contiguous.

Drivers should allocate memory from pool `type M_IOSYS` or `M_DMA`. The `M_IOSYS` pool is for general I/O purposes; the `M_DMA` pool is for DMA purposes.

The `flags` parameter may optionally have the `M_NOWAIT` flag set. If `M_NOWAIT` is set and no memory is available from the requested pool type, the functions will return `NULL`. Without `M_NOWAIT` set, the caller can be blocked and made to wait for memory to become available.
The M_NOWAIT flag must be set if MALLOC() is:

- called in an interrupt context, or
- called while holding a spinlock

If M_NOWAIT is set, the caller must be prepared to handle the case where no kernel memory has been allocated.

M_WAITOK should be passed as the flags parameter if the caller is able to block and wait for memory to become available.

**RETURN VALUES**

MALLOC() returns the following values to the pointer space:

- <>NULL Virtual address of kernel memory allocated.
- NULL No kernel memory has been allocated. NULL is returned only when the M_NOWAIT flag is set; otherwise, the caller is blocked and waits until memory becomes available.

**CONSTRAINTS**

If the M_NOWAIT flag is not set:

- the caller must not be in an interrupt context, and
- no spinlocks can be held

**EXAMPLES**

```c
my_stuct_t * my_addr;

MALLOC (my_addr, my_struct_t *, sizeof(my_struct_t),
    M_IOSYS, M_NOWAIT);

if (!my_addr) {
    /*
        * Error!  No kernel memory currently available.
        */
}
```
SEE ALSO

FREE(KER2), kfree(KER2), kmalloc(KER2)
NAME
map_mem_to_host(KER2) – Map physical bus address to host virtual space

SYNOPSIS

#include <sys/wsio.h>

caddr_t map_mem_to_host (struct isc_table_type *isc,
                      caddr_t phys_addr, size_t size);

PARAMETERS

isc ISC pointer that corresponds to the interface card
associated with this memory.

phys_addr Base physical address for a range of memory space on
an I/O bus.

size Size of the memory range in bytes.

DESCRIPTION

The kernel function map_mem_to_host() maps physical bit address to a
host virtual address for cards with memory ranges. Hardware disables
 caching for all EISA data accesses, including EISA memory space.

map_mem_to_host() will return NULL if a request overlaps an existing
mapping.

RETURN VALUES

<>NULL Successful completion. The value is the host virtual
address for accessing the space specified by the
parameters.

NULL Error.

CONSTRAINTS
SEE ALSO

unmap_mem_from_host(KER2), kernel_iomap(KER2),
kernel_iounmap(KER2)
NAME

minor(KER2) – Extract the minor number from a device number.

SYNOPSIS

#include <sys/sysmacros.h>

#define minor(x) ((long)((x)&0xffffff))

PARAMETERS

x A dev_t device number.

DESCRIPTION

minor() is a macro that returns the minor number from a device number.

CONSTRAINTS

SEE ALSO

major(KER2)
NAME

minphys(KER2) – Limit the b_bcount field in a buf structure to the value MAXPHYS

SYNOPSIS

#include <sys/buf.h>

void minphys (struct buf *bp);

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

The minphys() kernel function compares bp->b_bcount against the value MAXPHYS defined in <sys/param.h>. If bp->b_bcount is greater than MAXPHYS, then bp->b_bcount is changed to MAXPHYS.

minphys() is passed as the mincnt parameter to physio(). In this way, physio() can break a large data transfer into multiple smaller transfers, each no greater than MAXPHYS bytes in length.

RETURN VALUES

None.

EXAMPLES

int mydriver_read(dev_t dev, struct uio * uio)
{
    return physio(mydriver_strategy, NULL, dev, B_READ,
                  minphys, uio);
}

CONSTRAINTS
SEE ALSO

physio(KER2), driver_minphys(WSIO_DRV)
NAME

msg_printf(KER2) – Write diagnostic information to the kernel message buffer.

SYNOPSIS

#include <sys/kern_svcs.h>

int msg_printf(const char * format, °);

PARAMETERS

format – A set of printing characters and limited conversion specifications, as defined in printf(3S).

DESCRIPTION

The msg_printf() kernel function is a scaled down version of the C library printf() routine (see printf(3S)).

msg_printf() writes diagnostic information to the kernel message buffer msgbuf only and not to the console.

The kernel msg_printf() routine can accept the following formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Printed Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>%; no argument</td>
</tr>
<tr>
<td>%b</td>
<td>Characters from string argument; can include backslash-escape codes</td>
</tr>
<tr>
<td>%c</td>
<td>Character from integer argument</td>
</tr>
<tr>
<td>%d</td>
<td>Signed decimal from integer argument</td>
</tr>
<tr>
<td>%lx</td>
<td>The argument x is long integer; x is one of x, d, u, or o</td>
</tr>
<tr>
<td>%o</td>
<td>Octal from integer argument</td>
</tr>
<tr>
<td>%s</td>
<td>Characters from string argument</td>
</tr>
<tr>
<td>%u</td>
<td>Unsigned decimal from integer argument</td>
</tr>
<tr>
<td>%x</td>
<td>Hexadecimal from integer argument</td>
</tr>
</tbody>
</table>
These formats are the same as in `print (3S)`. Other formats specified in `printf (3S)` are not supported.

**RETURN VALUES**

`msg_printf()` returns the length of the formatted string.

**CONSTRAINTS**

**SEE ALSO**

`printf(KER2), printf(3S), sprintf(KER2), sprintf(3S)`
NAME

owns_spinlock(KER2) – Test whether a spinlock is owned by the processor.

SYNOPSIS

#include <sys/spinlock.h>

int owns_spinlock (lock_t * lock);

PARAMETERS

lock Pointer to a lock_t structure.

DESCRIPTION

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread’s current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The owns_spinlock() kernel function tests whether a spinlock, pointed to by lock, is owned (locked) by the processor.

RETURN VALUES

1 The processor owns the spinlock.
0 The processor does not own the spinlock. It may be owned by another processor, or not owned by any processor.

CONSTRAINTS
EXAMPLE

```c
int got_spinlock_here = 0;

/*
 * We may have already acquired the spinlock before getting
 * here. Be sure to check that we do not own the spinlock
 * before acquiring the lock, otherwise, we may spin forever.
 */
if (!owns_spinlock(mydrv_lock)) {
    spinlock(mydrv_lock);
    got_spinlock_here = 1;
}

/*
 * Do work under the protection of the spinlock.
 */
...

/*
 * Release the spinlock if it was acquired here.
 */
if (got_spinlock_here) {
    spinunlock(mydrv_lock);
}
```

SEE ALSO

alloc_spinlock(KER2), cspinlock(KER2), dealloc_spinlock(KER2),
get_sleep_lock(KER2), spinlock(KER2), spinunlock(KER2)
NAME

p_pgrp(KER2) – Return the process group identifier for a process.

SYNOPSIS

#include <h/proc_iface.h>

pid_t p_pgrp (proc_t * procp);

PARAMETERS

procp Pointer to a proc_t structure.

DESCRIPTION

The p_pgrp() kernel function returns the process group identifier for a process.

The procp parameter is a pointer to a proc_t structure. The pointer for the current process is contained in u.u_procp.

RETURN VALUES

The p_pgrp() kernel function returns the process group identifier for a process.

CONSTRAINTS

EXAMPLES

#include <sys/user.h>
#include <sys/signal.h>
#include <h/proc_iface.h>

/*
 * Signal all processes in the current process group.
 */
gsignal(p_pgrp(u.u_procp),SIGIO);
SEE ALSO

gsignal(KER2)
NAME

panic(KER2) – Soft-crash the operating system

SYNOPSIS

```c
#include <sys/kern_svcs.h>

void panic (char * str);
```

PARAMETERS

`str` Pointer to a character string message. No format capability.

DESCRIPTION

The `panic()` kernel function prints `str` to the system console and halts the system.

`panic()` prints the processor status register, the program counter register, the trap type on processor exceptions, and part of the kernel stack.

RETURN VALUES

`panic()` does not return.

CONSTRAINTS

WARNINGS

`panic()` halts the system and may cause file system damage. This call should only be used to flag catastrophic and unrecoverable failures.
NAME

**physio(KER2)** – Perform unbuffered, physical I/O data transfers.

SYNOPSIS

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

int physio (int (*strat)(), struct buf *bp, int dev, int flag,
            void (*mincnt)(), struct uio *uiop);
```

PARAMETERS

- **strat**: Pointer to the driver strategy function.
- **bp**: Pointer to a *buf* structure.
- **dev**: Device number.
- **flag**: Read/write flag: B_READ or B_WRITE.
- **mincnt**: Function that limits maximum transfer length.
- **uiop**: Pointer to *uio* structure.

DESCRIPTION

The **physio()** kernel function performs unbuffered, physical I/O data transfers. It accepts a user I/O request specified by a *uio* structure, prepares the data pages for I/O, builds the associated *buf* structure, calls the specified driver **strat** function, and waits for the I/O to complete.

The **strat** parameter is a pointer to the driver strategy function. **physio()** calls **strat** to start the I/O transfer, then waits by calling **biowait()**. When the I/O transfer completes, **strat** calls **biodone()** to awaken the waiting thread. **strat** reports an error to **physio()** by setting B_ERROR in **bp->b_flags** and putting an **errno** value in **bp->b_error**. See **errno** values in `<sys/errno.h>`.

The **bp** parameter is a pointer to a *buf* structure allocated by the caller. If set to **NULL**, **physio()** allocates the *buf* structure for the caller. **physio()** deallocates the *buf* structure when the I/O request completes.

The **dev** parameter is the device number passed to the driver.
The *flag* parameter indicates the direction of the I/O transfer. *B_READ* is used to read data from the device into memory; *B_WRITE* is used to write data from memory to the device.

The *mincnt* parameter is a pointer to a function that limits the data transfer length. Typically, drivers pass the kernel supplied function `minphys()` as this parameter.

The *uiop* parameter is a pointer to the *uioc* structure that is passed to the driver. The *uioc* structure specifies the following:

*uioc_iov* Pointer to an *iov* structure that contains the base address *iov_base* and transfer length *iov_len* of the I/O request.

*uioc_iovcnt* Number of *iov* structures. If >1, *uioc_iov* points to an array of *iov* structures.

*uioc_offset* Offset into device.

*uioc_seg* Type of memory segment to transfer. If set to *UIOSEG_USER*, `physio()` must be called in the user context.

*uioc_resid* Number of bytes of data remaining to be transferred.

For each *iov* structure, `physio()` performs the following:

1. Validate the user has appropriate access permissions for the data pages specified by *iov*.

2. Set up the *buf* structure with the following information:
   - *b_dev* is set to the device number.
   - *b_error* is set to zero.
   - *b_flags* is set with `B_BUSY`, `B_PHYS` and `B_RAW`. If the flag parameter is *B_WRITE*, *B_WRITE* is set.
   - *b_un.b_addr* is set to *iov_base*.
   - *b_bcount* is set to *iov_len*. This value can be adjusted by `mincnt`.
   - *b_blkno* is set to the *DEV_BSIZE* block number corresponding to *uioc_offset*.
3. Call `mincnt` to adjust the transfer length, if too large. If adjusted, `physio()` will make multiple calls to `strat` until all the data specified by `iov_len` has been transformed (or an error occurs).

4. If `uio_seg` is not `UIOSEG_KERNEL`, lock down the data pages to be transferred.

5. If `uio_seg` is not `UIOSEG_KERNEL` and the driver has set `C_MAP_BUFFER_TO_KERNEL` in the `d_flags` field of its `drv_ops_t` structure, `physio()` maps the user buffer into kernel space. This additional overhead is performed for legacy drivers that do not know how to access user space in their strategy function.

6. Calls `strat` passing the `buf` structure pointer `bp` as a parameter, then wait for the I/O request to complete by calling `biowait()`.

7. The driver calls `biodone()` when the I/O request completes to awaken the thread waiting in `biowait()`.

8. After the I/O request has completed, data pages that have been locked down are unlocked.

9. The following fields in the `uio` structure are updated:

   - `uio_resid` is decreased by the transfer length
   - `uio_offset` is increased by the transfer length

After all `iovec` structures have been processed, the `buf` structure is released and `physio()` returns to the caller.

**RETURN VALUES**

0 Successful completion.

<>0 Error.

**CONSTRAINTS**

Must not be called in an interrupt context.

Must not be called while holding a spinlock.
EXAMPLES

```c
int
mydriver_read(dev_t dev, struct uio * uio)
{
    return physio(mydriver_strategy, NULL, dev, B_READ,
                  minphys, uio);
}
```

SEE ALSO

biodone(KER2), biowait(KER2), buf(KER4), minphys(KER2),
uio(KER4)
NAME

printf(KER2) – Kernel print routine

SYNOPSIS

#include <sys/kern_svcs.h>

int printf (const char * fmt, º);

PARAMETERS

fmt A set of printing characters and limited conversion
specifications, as defined in printf(3S).
° An argument corresponding to a format conversion
specification, as defined in printf(3S).

DESCRIPTION

The printf() kernel function is a scaled down version of the C library
printf() routine (see printf(3S)).
printf() writes diagnostic information to the console and into the
kernel message buffer msgbuf.

The kernel printf(), routine can accept the following formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Printed Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>%; no argument</td>
</tr>
<tr>
<td>%b</td>
<td>Characters from string argument; can include backslash-escape codes</td>
</tr>
<tr>
<td>%c</td>
<td>Character from integer argument</td>
</tr>
<tr>
<td>%d</td>
<td>Signed decimal from integer argument</td>
</tr>
<tr>
<td>%lx</td>
<td>The argument x is long integer; x is one of x, d, u, or o</td>
</tr>
<tr>
<td>%o</td>
<td>Octal from integer argument</td>
</tr>
<tr>
<td>%s</td>
<td>Characters from string argument</td>
</tr>
<tr>
<td>%u</td>
<td>Unsigned decimal from integer argument</td>
</tr>
</tbody>
</table>
Kernel Reference Pages

printf(KER2)

%\x

Hexadecimal from integer argument

These formats are the same as in printf(3S). Other formats specified in printf(3S) are not supported.

The kernel printf() routine is buffered. Therefore, console error messages may not be synchronous with the current kernel state.

RETURN VALUES

printf() returns the length of the formatted string.

CONSTRAINTS

SEE ALSO

sprintf(KER2), msg_printf(KER2), printf(3S)
privlbcopy(KER2) – Copy data from a source buffer to a destination buffer using global virtual addresses.

SYNOPSIS

```c
#include <sys/kern_svcs.h>

int privlbcopy (space_t from_sid, caddr_t from_addr,
                space_t to_sid, caddr_t to_addr, size_t n);
```

PARAMETERS

- `from_sid` Space ID of source buffer.
- `from_addr` Address of source buffer.
- `to_sid` Space ID of destination buffer.
- `to_addr` Address of destination buffer.
- `n` Number of bytes to copy.

DESCRIPTION

The `privlbcopy()` kernel function copies data from a source buffer to a destination buffer using global virtual addresses. Protection is disabled to allow copies from/to buffers that are not owned by the current process.

The `from_sid` and `from_addr` parameters are the space ID and address, respectively, of the source buffer.

The `to_sid` and `to_addr` parameters are the space ID and address, respectively, of the destination buffer.

The space ID (SID) and address are combined to form the global virtual address of the buffer. If the address is in kernel space, then `ldsid()` can be used to get the SID of the address.
Do not call privlbcopy() to copy data from/to user space without first calling physio(). physio() validates access to the user buffer, locks the corresponding data pages in memory, puts the user SID and address in the b_spaddr and b_un.b_addr fields of the buf structure. (b_spaddr and b_un.b_addr will contain a kernel space ID and address if the driver specifies C_MAP_BUFFER_TO_KERNEL in its drv_ops_t structure.)

RETURN VALUES

0 Successful completion.
<>0 Error.

CONSTRAINTS

EXAMPLES

/*
 * Copy data from a kernel buffer to a user buffer that
 * has been validated and locked by physio().
 */
(void)privlbcopy(ldsid(kern_addr), kern_addr,
 bp->b_spaddr, bp->b_un.b_addr, bp->b_bcount);

WARNINGS

privlbcopy() must not be used for transfers between memory and I/O space. The underlying routines make choices of the optimal transfer code, which will probably not be supported on either the I/O bus adapter or the attached interface card. Failure to heed this warning may result in data corruption, an HPMC (High Priority Machine Check), or a call to panic().

SEE ALSO

bcopy(KER2), copyin(KER2), copyout(KER2), ldsid(KER2)
physio(KER2)
NAME

psignal(KER2) – Send the specified signal to a process

SYNOPSIS

#include <h/proc_iface.h>

void psignal (proc_t *procp, int sig);

PARAMETERS

procp Pointer to a proc_t structure.
sig Signal number.

DESCRIPTION

The psignal() kernel function sends the specified sig to the process specified by procp.

The procp parameter is a pointer to a proc_t structure. The pointer for the current process is contained in u.u_procp.

Signal numbers are defined in <sys/signal.h>.

psignal() checks the signal mask of the process to determine if the specified signal is being blocked, ignored, or caught. If a process is blocking the signal, the signal is recorded as pending so the signal is not lost. If a process is ignoring the signal, the signal is not sent. If the process is catching the signal, that process is put on the run queue. For example, if the process has called sleep() with priority greater than PZERO and PCATCH set, psignal() will awaken the sleeping process.

RETURN VALUES

None.

CONSTRAINTS
EXAMPLES

```c
#include <sys/user.h>
#include <sys/signal.h>
#include <h/proc_iface.h>

/*
 * In the user context, save the u_procp.
 */
proc_t * my_procp = u.u_procp;
...

/*
 * In the ISR, signal the process.
 */
psignal(my_procp, SIGIO);
```

SEE ALSO

gsignal(KER2)
NAME
putc(KER2) – Manipulate characters on a clist

SYNOPSIS

#include <sys/clist.h>

int putc (int ch, struct clist *list);

PARAMETERS

ch Character to place on the clist.
list Pointer to a clist.

DESCRIPTION

The putc() kernel function puts the character ch on the clist list.

RETURN VALUES

0 Successful completion.
-1 Error (probably exhausted the available list structures).

CONSTRAINTS
EXAMPLES

void catq( from, to )
struct clist * from;
struct clist * to;

/* concatenate characters from one clist to another */
/* expects putc to succeed */
{
    int c;

    while ((c = getc(from)) >= 0)
        if(putc(c, to))
            exit(1);
}

SEE ALSO

getc(KER2), getcb(KER2), getcf(KER2), putcb(KER2), putcf(KER2)
NAME

putcb(KER2) – Manipulate cblocks on a clist

SYNOPSIS

#include <sys/clist.h>

void putcb (struct cblock * cb, struct clist * list);

PARAMETERS

  cb          Pointer to a cblock.
  list        Pointer to a clist.

DESCRIPTION

putcb() adds the cblock pointed to by cb onto the clist pointed to by list.

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

getc(KER2), getcf(KER2), putc(KER2), putcf(KER2)
NAME

putcf(KER2) – Manipulate a cblock on the cblock free list

SYNOPSIS

#include <sys/clist.h>

void putcf (struct cblock * cb);

PARAMETERS

cb Pointer to a cblock structure.

DESCRIPTION

putcf() returns the cblock pointed to by cb to the free list.

RETURN VALUES

None.

CONSTRAINTS

EXAMPLES

static int
mydev_open(dev_t dev, int flag)
{
    struct cblock *cp;
    struct mydevice dv;

    /* Flush out the device buffer */
    while ((cp = getcb(&dv->queue)) != NULL)
        putcf(cp);
    ...
}
SEE ALSO

getcf(KER2), getc(KER2), getcb(KER2), putc(KER2), putcb(KER2)
NAME

seltrue(KER2) – Select driver entry point that returns true for traditional select flags.

SYNOPSIS

int seltrue (dev_t dev, int flag);

PARAMETERS

dev Device number.

flag Device select flags.

DESCRIPTION

The seltrue() kernel function may be used as the driver entry point for the select() system call. It returns true for the traditional select flag values of FREAD, FWRITE, and 0.

RETURN VALUES

seltrue() returns the following values:

0 Device select flag is not a traditional select option.

-1 Device select flag is either FREAD, FWRITE, or 0.

CONSTRAINTS
EXAMPLES

drv_ops_t lpr0_drv_ops = { /* driver entry points */
    lpr0_open, /* open */
    lpr0_close, /* close */
    NULL, /* strategy */
    NULL, /* dump */
    NULL, /* psize */
    NULL, /* reserved */
    lpr0_read, /* read */
    lpr0_write, /* write */
    lpr0_ioctl, /* ioctl */
    seltrue, /* select */
    NULL, /* option1 */
    NULL, NULL, NULL, NULL, /* reserved entry points */
    0, /* device flags */
};

SEE ALSO

select(2)
NAME
 selwakeup(KER2) – Wake up a kernel thread sleeping on a select condition

SYNOPSIS
 void selwakeup (struct kthread * threadp, int collision);

PARAMETERS
 threadp Pointer to the kernel thread to be awakened.
collision Indicates another kernel thread is sleeping on a select condition.

DESCRIPTION
 The selwakeup() kernel function wakes up a kernel thread sleeping on a select condition. If collision is zero, only the kernel thread pointed to by threadp is awakened. If collision is non-zero, all kernel threads sleeping on a select condition are awakened.

RETURN VALUES
 None.

CONSTRAINTS
 Must not be called while holding a spinlock of order >= SCHED_LOCK_ORDER.
EXAMPLE

#define MYSEL_COLL 1    /* my select collision flag */

static int
mydev_select(dev_t dev, int rw)
{
    struct kthread * kthread;
    ...
    /*
     * Check if another kernel thread is already
     * sleeping on a select condition with the driver.
     */
    kthread = myselstruct->thread;
    if (kthreadp &
        waiting_in_select (kthreadp) {
        * There is going to be a collision of multiple
        * threads sleeping on a select condition.
        */
        myselstruct->selflag |= MYSEL_COLLISION;
    } else {
        /*
         * Save the kernel thread pointer of this
         * thread that will sleep on a select condition.
         */
        myselstruct->thread = u.u_kthreadp;
    }
    ...
}

static void
mydev_selwakeup(void)
{
    selwakeup(myselstruct->thread,
        myselstruct->selflag & MYSEL_COLLISION);
    myselstruct->thread = NULL;
    myselstruct->selflag &= ~MYSEL_COLLISION;
}

SEE ALSO

waiting_in_select (KER2)
NAME

sleep(KER2) – Sleep on a channel

SYNOPSIS

#include <sys/kern_svcs.h>
#include <sys/param.h>

int sleep (void * chan, int pri);

PARAMETERS

chan Channel (i.e., kernel address) to sleep on.
pri Sleep priority.

DESCRIPTION

The sleep() kernel function sleeps on a channel chan and waits for a corresponding call to wakeup(). When awakened, the sleeping thread is scheduled at priority pri.

The chan parameter is typically a unique kernel address, such as a pointer to a private data area. This provides a one-to-one correspondence between wakeup events and chan. chan must not be zero.

The pri parameter specifies a value related to the system priority level of the sleeping thread. Block I/O drivers should set pri to the value PRIBIO, which is below the signal threshold value PZERO. If pri is set to a value greater than PZERO and the PCATCH flag is set (ORed with pri), the sleeping thread may be interrupted and awakened by a signal. If pri is greater than PZERO and PCATCH is not set, a signal handler may be invoked, and the call to sleep() may never return to the caller.

If more than one thread sleeps on chan, there is a race as to which thread wakes up first. All threads waiting for an event are awakened together. Each thread that returns from its call to sleep() should verify it is to handle the event; threads not handling the event may need to call sleep() again.

Prior to calling sleep(), a call to get_sleep_lock() may be required to protect against a race condition with wakeup(). The typical sequence executed by drivers is to:
Acquire the sleep lock.

Start an asynchronous activity

Sleep and wait for the asynchronous activity to complete.

The sleep lock is released by sleep() after the thread has been put on the sleep queue. This ensures that the thread will not miss the corresponding wakeup() which also acquires the same sleep lock.

Acquiring the sleep lock is not required if there is no race condition with wakeup(). For example, consider a driver set up to call wakeup() at regular intervals in a callout function specified by timeout(). A call to sleep() may miss a corresponding call to wakeup(), but the thread will be awakened by the next call to wakeup(), which occurs at regular intervals.

RETURN VALUES

0       Awakened by a corresponding call to wakeup().
1       Awakened by a signal if the PCATCH flag is set.

CONSTRAINTS

Must not be called in an interrupt context.
Must not be called while holding a spinlock.

EXAMPLE

/*
 * The sleeping thread waits on a channel by first calling
 * get_sleep_lock(), then it starts an asynchronous activity
 * and finally calls sleep() which releases the sleep lock.
 */
(void)get_sleep_lock(wait_chan);
start_async_activity();
(void)sleep(wait_chan, PRIBIO);
...
/*
 * When the asynchronous activity completes (typically in an
 * interrupt service routine), the sleeping thread is awakened.
 */
(void)wakeup(wait_chan);
SEE ALSO

get_sleep_lock(KER2), wakeup(KER2),
NAME

spinlock(KER2) – Acquire (lock) a spinlock

SYNOPSIS

```c
#include <sys/spinlock.h>

void spinlock (lock_t * lock);
```

PARAMETERS

- `lock`: Pointer to a `lock_t` structure.

DESCRIPTION

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread’s current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The `spinlock()` kernel function attempts to acquire (lock) the spinlock pointed to by `lock`. The processor spins and waits if the spinlock is currently locked.

Observe the following restrictions while holding a spinlock:

- Do not hold a spinlock for more than a few milliseconds.
- Do not acquire another spinlock of lower or equal lock order.
- Do not call an interface that can potentially block (i.e., sleep).

Spinlocks can be acquired while executing in an interrupt context as well as a user context. As such, they are able to synchronize the top and bottom halves of a driver.

RETURN VALUES

None
CONSTRAINTS

EXAMPLE

/*
 * Acquire the spinlock for my driver.
 */
spinlock(mydrv_lock);

/*
 * Manipulate driver data protected by the spinlock.
 * Do as little as possible here since external interrupts
 * to the processor are disabled while a spinlock is held.
 */
...

/*
 * Release the spinlock when done.
 */
spinunlock(mydrv_lock);

SEE ALSO

alloc_spinlock(KER2), cspinlock(KER2), dealloc_spinlock(KER2),
get_sleep_lock(KER2), owns_spinlock(KER2), spinunlock(KER2)
NAME

spinunlock(KER2) – Release (unlock) a spinlock.

SYNOPSIS

#include <sys/spinlock.h>

void spinunlock (lock_t * lock)

PARAMETERS

lock Pointer to a lock_t structure.

DESCRIPTION

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread’s current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The spinunlock() kernel function releases (unlocks) the spinlock pointed to by lock.

RETURN VALUES

None

CONSTRAINTS
EXAMPLE

/*
 * Acquire the spinlock for my driver.
 */
spinlock(mydrv_lock);

/*
 * Manipulate driver data protected by the spinlock.
 * Do as little as possible here since external interrupts
 * to the processor are disabled while a spinlock is held.
 */
...

/*
 * Release the spinlock when done.
 */
spinunlock(mydrv_lock);

SEE ALSO

alloc_spinlock(KER2), cspinlock(KER2), dealloc_spinlock(KER2),
get_sleep_lock(KER2), owns_spinlock(KER2), spinlock(KER2)
NAME

`sprintf(KER2)` – Kernel version of `sprintf()`.

SYNOPSIS

```c
#include <sys/kern_svc.h>

int sprintf(char * str, int len, const char * fmt, ...);
```

PARAMETERS

- `str` Address of buffer to hold the formatted string.
- `len` Length of the formatted string in bytes.
- `fmt` A set of printing characters and limited conversion specifications, as defined in `sprintf(3S)`.
- `...` An argument corresponding to a format conversion specification, as defined in `sprintf(3S)`.

DESCRIPTION

The `sprintf()` kernel function is a scaled down version of the C library `sprintf()` routine (see `sprintf(3S)`). However, the kernel version adds `len` as the second parameter.

`sprintf()` writes information to the `str` array.

The kernel `sprintf()`, routine can accept the following formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Printed Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>%; no argument</td>
</tr>
<tr>
<td>%b</td>
<td>Characters from string argument; can include backslash-escape codes</td>
</tr>
<tr>
<td>%c</td>
<td>Character from integer argument</td>
</tr>
<tr>
<td>%d</td>
<td>Signed decimal from integer argument</td>
</tr>
<tr>
<td>%lx</td>
<td>The argument x is long integer; x is one of x, d, u, or o.</td>
</tr>
<tr>
<td>%o</td>
<td>Octal from integer argument</td>
</tr>
</tbody>
</table>
Kernel Reference Pages

Kernel Reference Pages

KERNELS

sprintf(KER2)

% s Characters from string argument
% u Unsigned decimal from integer argument
% x Hexadecimal from integer argument

These formats are the same as in sprintf(3S). Other formats specified in
sprintf(3S) are not supported.

RETURN VALUES

sprintf() returns the length of formatted string.

CONSTRAINTS

SEE ALSO

msg_printf(KER2), printf(KER2), sprintf(3S)
NAME
strcat(KER2) – Concatenate two strings

SYNOPSIS

#include <sys/kern_svcs.h>

char * strcat (char * buf, const char * str);

PARAMETERS

buf Pointer to a string.
str Pointer to a string.

DESCRIPTION

The strcat() kernel function appends string buf to string char.

RETURN VALUES

strcat() returns buf.

CONSTRAINTS

SEE ALSO

strncpy(KER2)
NAME

`strcmp(KER2)` – Compare two strings

SYNOPSIS

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

int strcmp (const char * s1, const char * s2);
```

PARAMETERS

- `s1` Pointer to a character string.
- `s2` Pointer to a character string.

DESCRIPTION

The `strcmp()` kernel function compares two null-terminated strings. If the strings are the same, zero is returned. If the strings are different, the integer value of `(*s1-*s2)` is returned.

RETURN VALUES

- `0` The strings are identical.
- `<>0` The strings are different.

CONSTRAINTS
EXAMPLES

char *
scsi_decode_opcode(dev_type, op)
int dev_type;
int op;
{
    struct scsi_opcode *entry = scsi_opcode[op];
    int dev_type_bit = 1 << dev_type;
    int i;

    static int initialized;

    /*
     ** Do a sanity check on the scsi_opcode table.
     */
    if (!initialized)
    {
        i = strcmp(scsi_opcode[CMDread_element_status][0].op_str,
                    "Read Element Status");
        if (i == 0)
            initialized = 1;
        else
            return "BadTable";
    }
    .
    .
}

SEE ALSO

strcpy(KER2), strlen(KER2), strncmp(KER2), strncpy(KER2)
NAME

strcpy(KER2) – Copy the characters from one string to another string

SYNOPSIS

#include <sys/kern_svcs.h>

char * strcpy (char * s1, const char * s2);

PARAMETERS

s1 Pointer to the destination string.

s2 Pointer to the source string.

DESCRIPTION

The strcpy() kernel function copies the characters from the string pointed to by s2 to the string pointed to by s1, terminating at the first null character in s2. A pointer to the string s1 is returned. No bounds checking is done.

RETURN VALUES

strcpy() returns the string pointed to by s1.

CONSTRAINTS
EXAMPLES

{
    .
    .
    if ( slot_is_isa )
        strcpy(&desc[0],"ISA card ");
    else {
        strcpy(desc,"EISA card ");
        cvt_eisa_id_to_ascii( valid_card_id, id_str);
        /* id_str returns 8 bytes with last byte null */
        strcpy(&(desc[strlen(desc)]), id_str);
    }
    .
    .
}

SEE ALSO

strcmp(KER2), strlen(KER2), strncmp(KER2), strncpy(KER2)
NAME

(strlen(KER2)) – Gets the number of nonnull bytes in a string

SYNOPSIS

```c
#include <sys/kern_svcs.h>

int strlen (char * s);
```

PARAMETERS

`s` Pointer to the string.

DESCRIPTION

The `strlen()` kernel function returns the number of bytes in the string, up to but not including the first null.

RETURN VALUES

`strlen()` returns the number of nonnull bytes in the string.

CONSTRAINTS
EXAMPLES

/*
 * Hex dump a chunk of data with header into the dmesg buffer.
 */
msg_dump(label, buf, n)
char *label;
u_char *buf;
int n;
{
    int i, j, indent;

    msg_printf("	%s:", label);
    indent = strlen(label) + 1;

    for (i = 0; i < n; i++)
    {
        if ((i & 0xf) == 0 & i != 0)
        {
            msg_printf("\n\t");
            for (j = 0; j < indent; j++)
                msg_printf(" ");
            msg_printf(" %02x", buf[i]);
        }
        msg_printf("\n");
    }
}

SEE ALSO

strcmp(KER2), strcpy(KER2), strncmp(KER2), strncpy(KER2)
NAME

strncmp(KER2) – Compare the first $n$ characters of two strings

SYNOPSIS

#include <sys/kern_svcs.h>

int strncmp ( char * s1, char * s2, int n);

PARAMETERS

$s1$ Pointer to a string.

$s2$ Pointer to a string.

$n$ Number of bytes to compare.

DESCRIPTION

The `strncmp()` kernel function compares the first $n$ characters of two null-terminated strings. If the strings are the same, zero is returned. If the strings are different, the integer value of ($s1 - s2$) is returned.

RETURN VALUES

0 The strings are equivalent.

<>0 The strings are different.

CONSTRAINTS
EXAMPLES

{
  .
  .
  if ((strncmp(utsname.machine, "9000/720", 8) != 0) &&
      (strncmp(utsname.machine, "9000/730", 8) != 0) &&
      (strncmp(utsname.machine, "9000/750", 8) != 0)) {
    /*
     * This isn't a 720, 730 or 750. If there was an
     * audio card
     * it would have been found by pdc at boot time.
     */
    return((struct audio_descriptor *) 0);
  }
  .
  .
}

SEE ALSO

strcmp(KER2), strlen(KER2), strncmp(KER2), strncpy(KER2)
NAME

`strncpy(KER2)` – Copy characters between strings

SYNOPSIS

```c
#include <sys/kern_svcs.h>

char * strncpy(char * s1, char * s2, int n);
```

PARAMETERS

- `s1` Pointer to a string.
- `s2` Pointer to a string.
- `n` Number of bytes to copy.

DESCRIPTION

The `strncpy()` kernel function copies the characters from the string pointed to by `s2` to the string pointed to by `s1`. Copying terminates at `n` bytes or after a NULL character which ever occurs first. No bounds checking is done.

RETURN VALUES

A pointer to the string `s1` is returned. `strncpy()` returns a pointer to the copied (`s1`) string.

CONSTRAINTS
EXAMPLES

{
    .
    .
    /*
    ** Get the first four bytes of the inquiry data
    ** for dev_id and the vendor and product id for desc
    */
    strncpy(desc, iqr_data->inq2.vendor_id, 8);
    strncpy(&desc[8], iqr_data->inq2.product_id, 16);
    desc[24] = '\0';
    .
    .
    }

SEE ALSO

string(3C), strcmp(KER2), strlen(KER2), strncmp(KER2), strcpy(KER2)
NAME

\texttt{suser(KER2)} – Test if the current user is a superuser

SYNOPSIS

\begin{verbatim}
#include <sys/kern_svcs.h>

int suser (void);
\end{verbatim}

PARAMETERS

None.

DESCRIPTION

The \texttt{suser()} kernel function tests if the current user is a superuser.

RETURN VALUES

\begin{align*}
0 & \quad \text{The current user is a superuser.} \\
-1 & \quad \text{The current user is not a superuser.}
\end{align*}

CONSTRAINTS

Must be called in the user context.

SEE ALSO
NAME

sw_trigger(KER2) – Request a software trigger

SYNOPSIS

#include <sys/timeout.h>

void sw_trigger (struct sw_intloc *intloc, int (*proc)(char *), char * arg, int level, int sublevel);

PARAMETERS

intloc A pointer to a sw_intloc structure to be added to the queue of software triggers. The driver allocates the structure, zero-filled. The sw_trigger() routine initializes its fields.

proc The address of a routine to be called when the software trigger is executed.

arg The argument to be passed to proc.

level The priority level of the software trigger.

The level value has the following restrictions:

❑ Your driver cannot set a software trigger higher than your current processor priority level.

❑ You can not call sw_trigger() with level set to 7.

sublevel Currently, sublevels are not implemented. Drivers can safely use 0 as the last argument.

DESCRIPTION

The software trigger mechanism provides software triggering of interrupt service routines. The sw_trigger() routine arranges the calling of a routine in interrupt context at a given priority level.

❑ Your timeout routine can set up a software trigger so that it defers its timeout processing from level 5 to a lower level.
Use a software trigger when your driver needs to acknowledge a device's interrupt quickly, at a high level, but can do the rest of the interrupt processing less urgently, at a lower level.

Software triggers provide a way for the top half of a driver to trigger the lower half to perform a specific function.

The kernel uses a linked list of structures to represent software triggers waiting to be serviced. The kernel checks this list each time it finishes servicing an interrupt. Elements of the list are sw_intloc structures, defined in /usr/include/sys/timeout.h.

When it checks the list, the kernel processes all requests for software triggers whose level is greater than the current interrupt level. The kernel processes pending requests in decreasing order of priority.

The sw_trigger() routine checks to see if the structure to which intloc points is already on the trigger queue. If it is, the kernel throws this request away, thus permitting only one pending request per sw_intloc structure. If your driver needs to have more than one software trigger pending, it must use separate sw_intloc structures.

RETURN VALUES

sw_trigger() is a void function.

CONSTRAINTS

EXAMPLES

The following fragment of a skeleton driver acknowledges an interrupt from a card at a high priority, and then uses a software trigger to defer the bulk of the interrupt processing to a lower priority.
Kernel Reference Pages

sw_trigger(KER2)

#include <sys/types.h>
#include <sys/timeout.h>
struct sw_intloc mycard_intloc;

mycard_isr()
{
    int reason;
    /* stop card from interrupting */
    mycard->control = .....;

    /* determine reason for interrupt and do *
    * any immediate interrupt processing */
    reason = ...; /* values from card regs */

    /* set up sw_trigger() request to perform *
    * remainder of interrupt processing at *
    * a lower level */

    sw_trigger (<exc>&mycard_intloc,mycard_isrII,reason,3,0);

    return(0);
}

mycard_isrII( reason )
int reason; /* reason for interrupt, */
    /* passed by mycard_isr() */
{
    /* complete secondary interrupt processing */
    switch(reason)
    {
    case IOCOMPLETE:
        /* process I/O complete condition */
        case IOERROR:
            /* processing for I/O error */
            .
            .
    }
}

SEE ALSO
timeout(KER2) – Execute a callout function after a specified length of time.

SYNOPSIS

```
#include <sys/param.h>
#include <sys/callout.h>

callout_t * timeout(int (*func)(), caddr_t arg, int t);
```

PARAMETERS

- **func**: Function to call when the time value `t` expires.
- **arg**: Argument passed to the callout function `func`.
- **t**: Time value in number of clock ticks.

DESCRIPTION

The `timeout()` kernel function executes the specified callout function `func` after `t` clock ticks have expired. Execution of `func` takes place in an interrupt context at priority level 2 where external interrupts to the processor are enabled.

The `func` parameter is a pointer to a function that takes one argument. Although the prototype declares the function to return an int value, the kernel does not make use of the return value.

The `arg` parameter is passed as the one argument to `func`.

The `t` parameter specifies the number of clock ticks to wait before calling `func`. To express time in seconds, multiply `t` by `HZ`, where `HZ` is defined as the number of clock ticks per second in `<sys/param.h>`.

The call to `timeout()` returns immediately without waiting for the time value `t` to expire. The timeout can be cancelled by making a corresponding call to `untimeout()`.

RETURN VALUES

`timeout()` returns a pointer to a callout structure.
Kernel Reference Pages

**timeout(KER2)**

## CONSTRAINTS

Must not be called while holding a spinlock of order >= 
CALLOUT_LOCK_ORDER.

## WARNINGS

Callout resources are not dynamically expandable. Each call to 
`timeout()` allocates a callout resource, and the resource is not released 
until the time value expires or the timeout is cancelled. The kernel may 
panic if no callout resources are available.

## EXAMPLES

```c
/*
 * Set a timeout to call my_timeout_func() passing my_arg 
 * after 5 seconds have expired.
 */
(void)timeout(my_timeout_func, my_arg, 5*HZ);
...

static int
my_timeout_func(caddr_t arg)
{
    ...
}
```

## SEE ALSO

`Ktimeout(KER2), untimeout(KER2)`
NAME

uio(KER4) – Data descriptor for scatter/gather I/O requests.

SYNOPSIS

#include <sys/uio.h>

DESCRIPTION

The uio structure specifies an I/O request that can be fragmented into multiple buffers (scatter/gather I/O). uio contains a pointer to an array of iovec structures, each specifying the base address and length of a buffer. Buffers may be in either user space or kernel space.

For read() and write() system calls, the kernel allocates and fills out a uio structure pointing to a single iovec structure. The uio structure for readv() and writev() system calls points to an array of iovec structures. The uio structure is passed to the driver_read() or driver_write() entry point of the target driver.

Drivers should not access fields in the uio structure directly. The kernel function physio() is called to perform raw, unbuffered I/O and uiomove() is called to copy the specified buffer(s) to/from a kernel buffer. Fields in the uio structure are updated by physio() and uiomove().

STRUCTURE MEMBERS

The uio structure is defined in <sys/uio.h>. It has the following fields.

Table 2-3 The uio Structure Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct iovec *</td>
<td>uio_iov</td>
</tr>
<tr>
<td>size_t</td>
<td>uio_iovcnt</td>
</tr>
<tr>
<td>uint32_t</td>
<td>uio_seg</td>
</tr>
<tr>
<td>long</td>
<td>uio_resid</td>
</tr>
<tr>
<td>uint32_t</td>
<td>uio_fpflags</td>
</tr>
<tr>
<td>off_t</td>
<td>uio_offset</td>
</tr>
</tbody>
</table>
**Kernel Reference Pages**

**uio[KER4]**

**uio_iow**
Pointer to an array of iovec structures. Each iovec structure contains the base address and length of a buffer.

**uio_iovcnt**
Number of entries in the array of iovec structures.

**uio_seg**
Indicates the addresses of the buffers are in user space (UIOSEG_USER) or kernel space (UIOSEG_KERNEL).

**uio_resid**
Number of bytes remaining to be transferred. Initially, this equals the sum of all the buffer lengths.

**uio_fpflags**
Flags that indicate whether the driver should not wait:

- **FNDELAY**
  The driver should not wait if the requested data transfer can not be done immediately. The request should be terminated without returning an error code.

- **FNBLOCK**
  The driver should not wait if the requested data transfer can not be done immediately. The request should be terminated and return EAGAIN as the error code.

**uio_offset**
Starting logical byte address on the device where the data transfer is to occur. Applicability of this field to the driver is device dependent. **uio_offset** is usually applicable only to devices capable of seeking.

**SEE ALSO**

iovec[KER4], open(2), physio[KER2], read(2), readv(2), uiomove[KER2], write(2), writev(2)
NAME

uiomove(KER2) – Copy data between memory in kernel space and the space specified by a uio structure.

SYNOPSIS

#include <sys/uio.h>

int uiomove(caddr_t addr, int n, int flag, struct uio * uiop);

PARAMETERS

addr Address of kernel memory.
n Number of bytes to copy
flag Copy direction: UIO_READ, UIO_WRITE
uiop Pointer to a uio structure.

DESCRIPTION

The uiomove() kernel function copies data between memory in kernel space and the space specified by the uio structure pointed to by uiop. uiomove() is typically used to copy data between user space and kernel space.

The addr parameter is the address of kernel memory to/from which data are to be copied. Kernel memory can be allocated by calling kmalloc() or geteblk(). Do not allocate kernel memory on the stack if the allocation size is greater than 128 bytes, as this can potentially lead to a stack overflow.

The n parameter is the number of bytes to copy. Typically, n is set to the value in uiop->uio_resid. If this value is greater than the size of allocated kernel memory, multiple calls to uiomove() with a smaller value of n will be required.

The flag parameter determines the direction of the copy: UIO_READ copies data from addr, UIO_WRITE copies data to addr.
The `uiop` parameter is a pointer to the `uio` structure. If the `uio_segflag` field in the `uio` structure is `UIO_USERSPACE`, user space is to be accessed and `uiomove()` must be called in the user context. If `uio_seg` is `UIO_SYSSPACE`, kernel space is to be accessed and `uiomove()` can be called in any user context or in an interrupt context.

The `uio_iov` field in the `uio` structure points to an `iovec` structure. If `uio_iovcnt` is greater than 1, `uio_iov` points to an array of `iovec` structures. `uiomove()` walks through the array of `iovec` structures as it performs the copy.

If the copy is successfully completed, fields in the `uio` structure are updated as follows:

- `uio_offset` is increased by `n`.
- `uio_resid` is decreased by `n`.

For a single `iovec` structure (`uio_iovcnt = 1`), a successful completion will update the `iovec` as follows:

- `uio_iov->iov_base` is increased by `n`.
- `uio_iov->iov_len` is decreased by `n`.

For an array of `iovec` structures (`uio_iovcnt > 1`), a successful completion will increment each `iov_base` field appropriately and decrement each `iov_len` field appropriately.

**RETURN VALUES**

- 0  Successful completion.
- <>0  Error.

**CONSTRAINTS**

If the `uio_segflag` field in the `uio` structure is `UIO_USERSPACE`, user space is to be accessed and `uiomove()` must be called in the user context. If `uio_seg` is `UIO_SYSSPACE`, kernel space is to be accessed and `uiomove()` can be called in any user context or in an interrupt context.
EXAMPLES

```c
static int
mydriver_write(dev_t dev, struct uio * uiop)
{
    struct buf * bp;
    int count = uiop->uio iov->iov len;

    bp = geteblk(count);
    (void)uiomove(bp->b_un.b_addr, count, UIO_WRITE, uio);
    brelse(bp);
    ...
}
```

SEE ALSO

brelse(KER2), geteblk(KER2)
NAME

unmap_mem_from_host(KER2) – Unmap physical bus from host virtual space

SYNOPSIS

#include <sys/wsio.h>

int unmap_mem_from_host (struct isc_table_type *isc,  
caddr_t virt_addr, size_t size);

PARAMETERS

isc The ISC pointer used in the previous 
map_mem_to_host() call.

virt_addr The virtual address previously obtained with a 
map_mem_to_host() call,

size The size of the memory space in bytes.

DESCRIPTION

The kernel routine unmap_mem_from_host() is used to unmap 
previously mapped virtual space. Hardware disables caching for all EISA 
data accesses, including EISA memory space.

RETURN VALUES

unmap_mem_from_host() returns an indeterminate value. Treat it as if 
it was a void function.

CONSTRAINTS

SEE ALSO

map_mem_to_host(KER2)
NAME
untimeout(KER2) – Cancel a previous timeout request

SYNOPSIS

#include <sys/callout.h>

int untimeout (int (*func)(), caddr_t arg);

PARAMETERS

func The func parameter from the previous timeout request.
arg The arg parameter from the previous request.

DESCRIPTION

The untimeout() kernel function cancels a previous timeout request made by a call to timeout() or Ktimeout(). The func and arg parameters are compared against the list of timeout requests waiting to expire. If a match is found, the timeout request is cancelled by taking it off the list and releasing allocated resources.

RETURN VALUES

≥ 0 Number of clock ticks remaining in the canceled timeout request.
-1 Timeout request not found. This is a normal return if the timeout request has already expired.

CONSTRAINTS

Must not be called while holding a spinlock of order ≥ CALLOUT_LOCK_ORDER.

SEE ALSO

Ktimeout(KER2), timeout(KER2)
NAME
user_iomap(KER2) – Map physical bus space to currently running process.

SYNOPSIS

#include <sys/iomap.h>

caddr_t user_iomap(caddr_t vaddr, caddr_t phys_addr,
                   pgcnt_t count);

PARAMETERS

vaddr NULL or equivalent to phys_addr
phys_addr The start of the host physical address (not the bus relative address) for an area of memory on an I/O bus (EISA, for example).
count The size of the memory space in pages (4 Kbyte pages).

DESCRIPTION

The kernel routine user_iomap() is used to map physical I/O bus space into the process address space of a currently running process. If this is the first mapping of this space, the mapping will be done with PROT_URW protection. If this is not the first mapping, this mapping will inherit the protection set by the first mapping of the space.

Multiple calls to user_iomap() can be issued by a driver for the same I/O address range for the same or other processes. At process exit time, any I/O regions mapped through the use of user_iomap() will be automatically unmapped. Note that if other drivers and/or user processes have the same space mapped, then the I/O space itself will only be unmapped upon the last kernel_iounmap() or user_iounmap() request.
RETURN VALUES

<>NULL Successful completion. The value is the host virtual address for accessing the space specified by the parameters.

NULL Error.

CONSTRAINTS

EXAMPLES

*  
** example maps 1 page of I/O space at  
** 0xf0000000 to driver and the currently  
** running process with user read/write  
** protection  
*/
#include <sys/iomap.h>
#include <sys/mman.h>
.
.
{  
  int sm;
  char byte;
  caddr_t mapped_addr;
  caddr_t user_mapped_addr;
  caddr_t phys_addr = 0xf0000000;
  pgcnt_t count = 1;
  
  /*
   ** n.b. must be in user context
   ** map the space to the driver
   ** kernel_iomap need not be done
   ** before user_iomap - they can
   ** be done in any order
   */
  mapped_addr =
    kernel_iomap(NULL,phys_addr,count,PROT_URW);
  if (mapped_addr == NULL)
    goto map_failed; /* bailout code for error */
  
  /*
** map in the same space to the
** currently running process
** user_mapped_addr should be
** directly returned to the process
*/
user_mapped_addr = user_iomap(NULL, phys_addr, count);

}

WARNINGS

Hardware disables caching for all EISA data accesses, including EISA memory space.

In the case of EISA (not ISA) cards, the EISA IO space is mapped (prior to calling the driver attach routine) with PROT_KRW. This mapping is for one page (4 Kbytes). If you desire user access to EISA IO space you will need to first unmap isc->if_reg_ptr with a call to unmap_mem_from_host(), and then remap the space with kernel_iomap_public(). A similar situation exists for PCI memory space.

user_iomap() will return NULL if a request overlaps an existing mapping. If the request exactly overlays (same start address and size) an existing mapping, the mapping services will return the host virtual address requested.

SEE ALSO

unmap_mem_from_host(KER2), user_iomap(KER2),
kernel_iounmap(KER2), user_iounmap(KER2)
NAME

user_iounmap(KER2) – Unmap physical bus from user process virtual space

SYNOPSIS

#include <sys/iomap.h>

int user_iounmap (caddr_t vaddr, u_int count);

PARAMETERS

vaddr The virtual address previously obtained with a
user_iomap() call,
count The size of the memory space in pages (4 KByte pages).

DESCRIPTION

The kernel routine user_iounmap() is used to unmap, from the user
process, previously mapped virtual space. If other drivers and/or user
processes have the same space mapped, then the I/O space will only be
unmapped upon the last kernel_iounmap() or user_iounmap() call.

RETURN VALUES

1 Error.
0 Success.

CONSTRAINTS

SEE ALSO

kernel_iomap(KER2), kernel_iomap_public(KER2),
kernel_iounmap(KER2), user_iomap(KER2)
NAME

VASSERT(KER2) – Test an assertion if OSDEBUG kernel

SYNOPSIS

#include <sys/debug.h>

VASSERT(expr);

PARAMETERS

expr An expression that evaluates to true or false.

DESCRIPTION

The VASSERT() kernel function calls panic() if the kernel was compiled with OSDEBUG on and if the expression expr is false. Otherwise, it does nothing.

RETURN VALUES

None

CONSTRAINTS

SEE ALSO

panic(KER2)
NAME

`waiting_in_select(KER2)` – Determines if a kernel thread is waiting on a select condition.

SYNOPSIS

```c
int waiting_in_select (struct kthread * threadp);
```

PARAMETERS

- `threadp` Pointer to a kernel thread.

DESCRIPTION

The `waiting_in_select()` kernel function determines if the specified kernel thread `threadp` is waiting on a select condition. This function replaces accesses to the global variable `selwait` which is no longer supported.

RETURN VALUES

`waiting_in_select()` returns the following values:

- 0 Thread is not waiting on a select condition.
- 1 Thread is waiting on a select condition.

CONSTRAINTS
EXAMPLES

#define MYSEL_COLLISION 1 /* my select collision flag */

static int
mydev_select(dev_t dev, int rw)
{
    struct kthread * kthreadp;
    ...
    /*
    * Check if another kernel thread is already
    * sleeping on a select condition with the driver.
    */
    kthreadp = myselstruct->thread;
    if (kthreadp &
        waiting_in_select(kthreadp) {
        /*
        * There is going to be a collision of multiple
        * threads sleeping on a select condition.
        */
        myselstruct->selflag |= MYSEL_COLLISION;
    } else {
        /*
        * Save the kernel thread pointer of this
        * thread that will sleep on a select condition.
        */
        myselstruct->thread = u.u_kthreadp;
    }
    ...
}

static void
mydev_selwakeup(void)
{
    selwakeup(myselstruct->thread,
        myselstruct->selflag & MYSEL_COLLISION);
    myselstruct->thread = NULL;
    myselstruct->selflag &= ~MYSEL_COLLISION;
}

SEE ALSO

selwakeup(KER2)
NAME

wakeup(KER2) – Wake up all threads sleeping on a channel.

SYNOPSIS

#include <sys/kern_svcs.h>

int wakeup (void * chan);

PARAMETERS

chan Channel passed in the corresponding call to sleep().

DESCRIPTION

The wakeup() kernel function wakes up all threads sleeping on a chan channel. Each thread sleeping on chan is scheduled with the priority specified when the thread called sleep().

The chan parameter is typically a unique kernel address, such as a pointer to a private data area. This provides a one-to-one correspondence between wakeup events and chan. chan must not be zero.

If more than one thread sleeps on chan, there is a race as to which thread wakes up first. All threads waiting for an event are awakened together. Each thread that returns from its call to sleep() should verify it is to handle the event; threads that do not handle the event may need to call sleep() again.

RETURN VALUES

wakeup() returns the number of threads awakened.

CONSTRAINTS

Must not be called while holding a spinlock of order greater than SLEEP_Q_LOCK_ORDER.
EXAMPLES

/*
 * The sleeping thread waits on a channel by first calling
 * get_sleep_lock(), then it starts an asynchronous activity
 * and finally calls sleep() which releases the sleep lock.
 */
(void) get_sleep_lock(wait_chan);
start_async_activity();
(void) sleep(wait_chan, PRIBIO);
...

/*
 * When the asynchronous activity completes (typically in an
 * interrupt service routine), the sleeping thread is awakened.
 */
(void) wakeup(wait_chan);

SEE ALSO

gsleep_lock(KER2) sleep(KER2)