Device Drivers

The Device Kit − Table of Contents

Device Drivers

Writing device drivers requires additional knowledge of the inner workings of the BeOS. To write a driver you must follow the rules laid out in this chapter very carefully. These rules are not the same as those for writing a normal applicationif your driver tries to do things it's not allowed to do, it could bring down the system.

This introduction covers how drivers interact with the kernel.

The Kernel and the Driver Author

The BeOS kernel comprises the basic functionality of the operating system: It knows how to start the boot process and to manage memory and threads, and it contains the PCI bus manager, the ISA bus manager, the device file system (**devfs**, which manages **/dev**), the root file system (**rootfs**, which manages **/**), and a few other things.

But this isn't enough to satisfy the needs of most applications, so the kernel uses add−ons to provide additional functionality. During the boot process, add−ons are loaded to handle "real" file systems, devices, busses, and the like.

Although Be's kernel add−ons provide support for a wide range of hardwarefrom disk devices to joysticksthis support isn't all−inclusive. Hardware developers may need to create their own drivers for their products.

Types of Kernel Add−on

There are three types of kernel add−on:

- *Device drivers* are add−ons that communicate directly with devices.
- *Modules* are kernel space add−ons that export an API for use by drivers (or by other modules).
- *File systems* are add−ons that support specific file systems, such as BFS, DOSFS, HFS, and so forth.

Device drivers and file systems, while extending the functionality of the kernel, are still accessible from user space: Applications can open and address them using file descriptors. Modules, on the other hand, are kernel−only units. Applications have no access to them; they're provided strictly for use by the kernel and other kernel add−ons.

Device Drivers

A device driver is an add−on that recognizes a specific device (or class of devices) and provides a means for the rest of the system to communicate with it. Usually this communication involves some form of device−specific protocol. For example, if the system wants to use an Ethernet card or graphics card, it needs to load a device driver add−on that knows how to communicate with that card. Similarly, code that knows how to talk to a class of devices (SCSI disks, ATA devices, ATAPI disks, or USB input devices, etc.) must be implemented as a device driver add−on.

Modules

Modules provide a uniform API for use by other modules and drivers. A module is like a library in that it acts as a repository for common code that's shared among several drivers.

For example: Let's say you have a device driver that talks to a SCSI device connected to a SCSI bus. A computer can have multiple SCSI busses. Because all SCSI devices use the same command set independent of the particular controller used to send the commands, the command set can be (and is) implemented as a module. The SCSI module knows how to handle all SCSI cards the BeOS supports; the API that the SCSI module defines is adopted by and augmented by the modules for specific SCSI device types (hard disks, scanners, CD drives, etc). The SCSI device modules are managed by a SCSI bus manager module, which knows how to cope with multiple busses and presents them in encapulated form to the drivers. The drivers then only need to deal with the bus manager's API, which makes the life of a driver author much simpler.

Be provides bus managers for SCSI, USB, IDE, and PCMCIA.

File Systems

File system add−ons provide support for disk and network file systems, such as BFS, HFS, FAT, ISO 9660, CIFS, and so forth. By creating new file system add−ons, developers can provide access to disks that are formatted using other file system.

Interactions with the Kernel

The kernel provides a number of services that drivers and modules can use. These include:

- Enabling and disabling interrupts.
- Setting up memory for DMA transactions.
- Access to other devices and modules.

The kernel also provides, at the user level, a Posix−like API for accessing devices. An application can open a device through **open()**, and use **read()**, **write()**, and **ioctl()** to access the device.

The Posix functions are converted into system calls into the kernel, which then passes them, via **devfs**, to the appropriate device driver.

---

devfs

The kernel manages device drivers through **devfs**, the device file system that's mounted at **/dev** during the boot process. In order to be accessed, a driver must "publish" itself by adding an entry in the **/dev** hierarchy. The basic Posix I/O functions (**open()**, **read()**, **write()**, **readv()**,

writev(), **ioctl()**, and **close()**) can then be used.

Devfs makes the drivers available as needed in **/dev**; this usually happens the first time a program iterates through the directory entries for a subdirectory in **/dev**. The kernel knows where in the **/dev** hierarchy to publish drivers based on their location in **/boot/beos/system/add−ons/kernel/drivers/dev**. For example, the ATAPI driver publishes drivers in **/dev/disk/ide/atapi**, the driver is located in **/boot/beos/system/add−ons/kernel/drivers/dev/disk/ide/atapi**. Whew.

You can see this device hierarchy by using the "ls" command from a Terminal window. "ls /dev" will show you the root of the device hierarchy, "ls /dev/disk" will show you disk device busses, "ls /dev/disk/ide" will show you the IDE devices, and so forth.

In reality, drivers tend to publish themselves in multiple locations in the **/dev** hierarchy, so instead of putting duplicate copies of the driver in the **.../drivers/dev** tree, the driver binaries are put at **/boot/beos/system/add−ons/kernel/drivers/bin**, and symlinks are created in the **.../drivers/dev** tree at the appropriate place. (The same is also done for drivers in **/boot/home/config/add−ons/kernel/drivers/...**.)

Driver Implementation Principles

Much of the stability of the BeOS is achieved by constructing a nearly impenetrable wall between the kernel and user applications. Drivers are chinks in that wall. If a driver misbehaves or fails, there's a strong possibility that it will cause unexpected behavior or kill the entire system. It's absolutely critical that drivers not only be very carefully tested before being released to the public, but that they follow the rules to the letter.

Kernel Space vs. User Space

One way you can reduce the risk of your driver causing a general system failure is by putting as much code as possible in user space. Create a driver that loads into kernel space just enough code to handle the low−level interactions that absolutely have to be done in kernel space, then load code into user space to handle the rest of the work. If the add−on fails, the system will keep runningonly your driver will fail.

Another plus to placing as much of your code as possible into user space is that it's much easier to debug code running in user space. Conventional debugging techniques that don't work for kernel code can be applied, and there's much less chance of taking down the system in the process.

Code Synchronization

Normally, spinlocks are a bad thing. A spinlock is a tight loop that watches for a condition to occur, looping endlessly until that condition is met (this is called "busy waiting"). This wastes valuable processor time, and is normally discouraged.

In general, you're encouraged to use semaphores instead of spinlocks; however, you can't acquire a semaphore while handling an interrupt. So if you need to synchronize code while handling an interrupt, you must use a spinlock. Put simply:

- Use spinlocks to protect critical sections in interrupt−handling code.
- Use semaphores in any other situation that calls for code synchronization.

Anywhere you use a spinlock to protect a critical section, you should disable interrupts. Of course, in an interrupt handler, you know that interrupts are already disabled, so you don't need to explicitly disable interrupts yourself. Interrupt handlers include I/O interrupts installed using **install_io_interrupt()** and timer interrupts installed by calling **add_timer()**.

Functions Available During Spinlocks

While your spinlock is running, you can perform the following actions. If it's not on this list, you can't do it.

- You can examine and alter hardware registers by using the appropriate bus manager hooks.
- You can examine and alter any locked−down memory.
- You can call the following kernel functions: **system_time()**, **atomic_add()**, **atomic_or()**, **atomic_and()**.
- You can call the following bus manager functions: **read_io_*()** and **write_io*()**.

If you do anything else inside your spinlock, you're breaking the rules, so don't do it.

Using Spinlocks

You need to be sure that your calls to **acquire_spinlock()** and **release_spinlock()** are balanced. In addition, if you nest spinlocks, they must be released in logical orderthat is, in the opposite order in which they're acquired.

The kernel keeps track of which spinlocks are being held and which are being waited upon. The kernel assumes that spinlocks are initialized to 0, and then acquired and released in logical order.

By keeping track of spinlocks, the kernel can detect and break deadlocks on multiprocessor systems.

Disabling Interrupts

The only time you should ever disable interrupts in a device driver is just before entering a spinlock−protected critical section. There is absolutely no other reason to do it, so don't.

After disabling interrupts, you should reenable them as quickly as possible. You must **never**, under any circumstances, leave interrupts disabled for more than 50 microseconds. This means that your interrupt handler code (which runs with interrupts implicitly disabled) must execute in 50 microseconds or less.

Functions Available While Interrupts Are Disabled

If you have interrupts disabled and aren't in a spinlock, you can do the following things in addition to those listed above in "Functions Available During Spinlocks":

- You can call **`release_sem_etc()`** with the **`B_DO_NOT_RESCHEDULE`** flag set.
- You can call **`get_sem_count()`**, **`add_timer()`**, **`cancel_timer()`**, and **`dprintf()`**.

If you feel that you need to call a function not explicitly listed as permitted here, please contact Be Developer Support at devsupport@be.com and explain your needs; we'd be happy to discuss the situation with you.

Don't Block

It's crucial that your interrupt handler never block, whether directly (by acquiring a semaphore, for example) or indirectly (by calling a function that might block).

Blocking can happen in a surprisingly large number of BeOS functions. It's obvious that **`acquire_sem()`** can block, but you might not be aware that functions such as **`malloc()`** or **`read_port()`** can block. Even touching unlocked memory areas can block because of virtual memory hits.

The point is this: If the BeOS function you want to call isn't explicitly listed in this section as one you can use, **don't call it**.

Don't Preempt

Your interrupt handler or spinlock section can't be preempted. Preemption could occur if you call **`release_sem()`** or **`release_sem_etc()`** without specifying the **`B_DO_NOT_RESCHEDULE`** flag. Normally, **`release_sem()`** lets the scheduler preempt your thread to allow other threads to acquire the semaphore as fast as possible. By specifying **`B_DO_NOT_RESCHEDULE`**, you tell the scheduler to allow your thread to continue running after it releases the semaphore.

If your interrupt handler wants to ensure that any preemption is handled immediately, it should specify **`B_DO_NOT_RESCHEDULE`** when calling **`release_sem()`**, then return **`B_INVOKE_SCHEDULER`**. This causes the scheduler to immediately handle preemption after your interrupt handler returns, instead of resuming the interrupted task. This is especially useful if your code called **`release_sem_etc()`** to release a semaphore that will allow other code to run elsewhere (such as in your driver's corresponding user−space code).

Again, when you call **`release_sem_etc()`**, be sure to specify the **`B_DO_NOT_RESCHEDULE`** flag to avoid any chance of preemption.

In summary, the order in which you should do things is this:

- Disable interrupts.
- Acquire the spinlock.
- Perform your tasks.
- Release the spinlock.
- Restore the original interrupt state.

File I/O

Sometimes a driver needs to be able to access disk files. Perhaps the driver has a preference file it needs to read. There are two ways to do this. You can use Posix I/O calls, or you can use the driver settings API provided by BeOS. The latter is preferred.

Using Posix Calls

Under BeOS, device drivers can access disk files using the standard low−level Posix I/O functions: **`open()`**, **`close()`**, **`read()`**, **`write()`**, and so forth. There aren't any special chores to attend to beforehand. Just **`open()`** the file and do your thing.

Two Posix extensions that might be helpful when you're writing code to perform file I/O from a device driver: **`readv()`** and **`writev()`**.

```
int readv(int fd, const struct iovec *vector, size_t count);

int writev(int fd, const struct iovec *vector, size_t count);

struct iovec {
   __ptr_t iov_base;
   size_t iov_len;
};
```

These functions provide a means to read and write contiguous portions of a file from multiple buffers. *vector* is a pointer to an array containing *count* vector records, each of which contains a pointer to a buffer, and the size of the buffer. **`readv()`** fills these buffers with data from the file, and **`writev()`** writes them to the file, in order.

When successful, **`readv()`** returns the number of bytes read.

For example, if your code needs to write two separate 1k buffers into a file, one after the other, you might do something like this:

```
struct iovec v[2];
v[0].iov_base = &buffer1;
v[0].iov_len = 1024;
v[1].iov_base = &buffer2;
```

```
v[1].iov_len = 1024;
if (writev(fd, &v, 2) != B_OK) {
   /* error */
}
```

Performing vectored I/O like this is often faster than doing multiple calls to **`read()`** and **`write()`**.

The Driver Settings API

If your driver is loaded before the file system for the disk on which your settings file resides, your driver might not be able to load its settings using Posix calls. The driver settings API lets you work around this circumstance. See the "Driver Settings API" section for details.

Writing Drivers

A device driver is an add−on that communicates with a specific device or type of device. Usually this communication involves some form of device−specific protocol. For example, an add−on that specifically addresses an Ethernet card or graphics card is a device driver. Likewise, add−ons that know how to talk to a class such as SCSI disks, ATA devices, ATAPI disks, or USB input devices is also a device driver.

A driver's job is to recognize the device and provide a means for applications to talk to it.

We can't stress this enough: a bug in a device driver can bring down the entire system. Be very careful, and be sure to test your work well.

To reduce the risk of the system being adversely affected by a bug in your code, you should put as much of your code into user space as possible.

This section covers the structure of device drivers, and provides some examples of how to write them.

Symbols Drivers Export

The kernel communicates with drivers by calling certain known entry points, which the driver must implement and export. These entry points are:

- **`init_hardware()`**
  Called when the system is booted, to let the driver detect and reset the hardware.
- **`init_driver()`**
  Called when the driver is loaded, so it can allocate needed system resources.
- **`uninit_driver()`**
  Called just before the driver is unloaded, so it can free allocated resources.
- **`publish_devices()`**
  Called to obtain a list of device names supported by the driver.
- **`find_device()`**
  Called to obtain a list of pointers to the hook functions for a specified device.
- **`api_version`**
  This exported value tells the kernel what version of the driver API it was written to, and should always be set to **`B_CUR_DRIVER_API_VERSION`** in your source code.

init_hardware()

```
status_t init_hardware(void)
```

This function is called when the system is booted, which lets the driver detect and reset the hardware it controls. The function should return **`B_OK`** if the initialization is successful; otherwise, an appropriate error code should be returned. If this function returns an error, the driver won't be used.

init_driver()

```
status_t init_driver(void)
```

Drivers are loaded and unloaded on an as−needed basis. When a driver is loaded by devfs, this function is called to let the driver allocate memory and other needed system resources. Return **`B_OK`** if initialization succeeds, otherwise return an appropriate error code. <<<what happens if this returns an error?>>>

uninit_driver()

```
void uninit_driver(void)
```

This function is called by devfs just before the driver is unloaded from memory. This lets the driver clean up after itself, freeing any resources it allocated.

publish_devices()

```
const char **publish_devices(void)
```

Devfs calls **publish_devices()** to learn the names, relative to /dev, of the devices the driver supports. The driver should return a **NULL**−terminated array of strings indicating all the installed devices the driver supports. For example, an ethernet device driver might return:

```
static char *devices[] = {
   "net/ether",
   NULL
};
```

In this case, devfs will then create the pseudo−file **/dev/net/ether**, through which all user applications can access the driver.

Since only one instance of the driver will be loaded, if support for multiple devices of the same type is desired, the driver must be capable of supporting them. If the driver senses (and supports) two ethernet cards, it might return:

```
static char *devices[] = {
   "net/ether1",
   "net/ether2",
   NULL
};
```

find_device()

```
device_hooks *find_device(const char *name)
```

When a device published by the driver is accessed, devfs communicates with it through a series of hook functions that handle the requests.The **find_device()** function is called to obtain a list of these hook functions, so that devfs can call them. The device_hooks structure returned lists out the hook functions.

The device_hooks structure, and what each hook does, is described in the next section.

api_version

```
int32 api_version;
```

This variable defines the API version to which the driver was written, and should be set to **B_CUR_DRIVER_API_VERSION** at compile time. The value of this variable will be changed with every revision to the driver API; the value with which your driver was compiled will tell devfs how it can communicate with the driver.

Device Hooks

The hook functions specified in the device_hooks function returned by the driver's **find_device()** function handle requests made by devfs (and through devfs, from user applications). These are described in this section.

The structure itself looks like this:

```
typedef struct {
   device_open_hook open;
   device_close_hook close;
   device_free_hook free;
   device_control_hook control;
   device_read_hook read;
   device_write_hook write;
   device_select_hook select;
   device_deselect_hook deselect;
   device_readv_hook readv;
   device_writev_hook writev;
} device_hooks;
```

In all cases, return **B_OK** if the operation is successfully completed, or an appropriate error code if not.

open_hook()

```
status_t open_hook(const char *name, uint32 flags, void **cookie)
```

This hook function is called when a program opens one of the devices supported by the driver. The name of the device (as returned by `publish_devices()`) is passed in name, along with the flags passed to the Posix **`open()`** function. *cookie* points to space large enough for you to store a single pointer. You can use this to store state information specific to the **`open()`** instance. If you need to track information on a per−**`open()`** basis, allocate the memory you need and store a pointer to it in **`*cookie`**.

---

close_hook()

```
status_t close_hook(void **cookie)
```

This hook is called when an open instance of the driver is closed using the **`close()`** Posix function. Note that because of the multithreaded nature of the BeOS, it's possible there may still be transactions pending, and you may receive more calls on the device. For that reason, you shouldn't free instance−wide system resources here. Instead, you should do this in `free_hook()`. However, if there are any blocked transactions pending, you should unblock them here.

---

free_hook()

```
status_t free_hook(void **cookie)
```

This hook is called once all pending transactions on an open (but closing) instance of your driver are completed. This is where your driver should release instance−wide system resources. **`free_hook()`** doesn't correspond to any Posix function.

---

read_hook()

```
status_t read_hook(void *cookie, off_t position, void *data, size_t *len)
```

This hook handles the Posix **`read()`** function for an open instance of your driver. Implement it to read *len* bytes of data starting at the specified byte *position* on the device, storing the read bytes at *data*. Exactly what this does is device−specific (disk devices would read from the specified offset on the disk, but a graphics driver might have some other interpretation of this request).

Before returning, you should set *len* to the actual number of bytes read into the buffer. Return `B_OK` if data was read (even if the number of returned bytes is less than requested), otherwise return an appropriate error.

---

readv_hook()

```
status_t readv_hook(void *cookie, off_t position, const struct iovec *vec,
    size_t count, size_t *len)
```

This hook handles the Posix **`readv()`** function for an open instance of your driver. This is a scatter/gather read function; given an array of iovec structures describing address/length pairs for a group of destination buffers, your implementation should fill each successive buffer with bytes, up to a total of *len* bytes. The *vec* array has *count* items in it.

As with `read_hook()`, set *len* to the actual number of bytes read, and return an appropriate result code.

---

write_hook()

```
status_t write_hook(void *cookie, off_t position, void *data, size_t len)
```

This hook handles the Posix **`write()`** function for an open instance of your driver. Implement it to write *len* bytes of data starting at the specified byte *position* on the device, from the buffer pointed to by *data*. Exactly what this does is device−specific (disk devices would write to the specified offset on the disk, but a graphics driver might have some other interpretation of this request).

Return `B_OK` if data was read (even if the number of returned bytes is less than requested), otherwise return an appropriate error.

---

writev_hook()

status_t **writev_hook(**void *_cookie_, off_t _position_, const struct iovec *_vec_, size_t _count_, size_t *_len_**)**

This hook handles the Posix **writev()** function for an open instance of your driver. This is a scatter/gather write function; given an array of iovec structures describing address/length pairs for a group of source buffers, your implementation should write each successive buffer to disk, up to a total of *len* bytes. The *vec* array has *count* items in it.

Before returning, set *len* to the actual number of bytes written, and return an appropriate result code.

control_hook()

status_t **control_hook(**void *_cookie_, uint32 _op_, void *_data_, size_t _len_**)**

This hook handles the **ioctl()** function for an open instance of your driver. The control hook provides a means to perform operations that don't map directly to either **read()** or **write()**. It receives the *cookie* for the open instance, plus the command code *op* and the *data* and *len* arguments specified by **ioctl()**'s caller. These arguments have no inherent relationship; they're simply arguments to **ioctl()** that are forwarded to your hook function. Their definitions are defined by the driver. Common command codes can be found in **be/drivers/Drivers.h**.

The *len* argument is only valid when **ioctl()** is called from user space; the kernel always sets it to 0.

select_hook() , deselect_hook()

status_t **select_hook(**void *_cookie_, uint8 _event_, uint32 _ref_, selectsync *_sync_**)**

status_t **deselect_hook(**void *_cookie_, uint8 _event_, uint32 _ref_, selectsync *_sync_**)**

These hooks are reserved for future use. Set the corresponding entries in your device_hooks structure to **NULL**.

Driver Rules

Keep the following rules in mind for each instance of your driver:

- **open()** will be called first, and no other hooks will be called until **open()** returns.
- **close()** may be called while other requests are pending. As previously mentioned, if you have blocked transactions, you must unblock them when **close()** is called. Further calls to other driver hooks my continue to occur after **close()** is called; however, you should return an error to any such requests.
- **free()** isn't called until all pending transactions for the open instance are completed.
- Multiple threads may be accessing the driver's hooks simultaneously, so be sure to lock and unlock where appropriate.

Writing Modules

Modules provide services that can be used by other modules, by device drivers, and by the kernel itself. They can be dynamically loaded and unloaded by the kernel, as needed. If a client can't find a module it needs, it will still load, which gives it the opportunity to find another way to perform the desired tasks, or to disable those features of itself.

Modules, like drivers, export an API through a structure that provides pointers to the functions provided by the module, along with other information about the module. You do this by expanding upon the basic module definition in **be/drivers/module.h**. For example, you might define your module information structure like this:

```
#define MY_MODULE_NAME "generic/mymodule/v1"

struct my_module_info {
   module_info module;
   int32 (*function1)();
   int32 (*function2)();
   void (*configure)(int32 parameter, int32 value);
};
```

Note that the first field in your module information structure is a module_info, which looks like this:

```
struct module_info {
   const char *name;
   uint32 flags;
   status_t (*std_ops);
};
```

The *name* field should be a pointer to the driver's name as indicated in your module's header file (in this example, **MY_MODULE_NAME**).

The *flags* field specifies which flags should be in effect for your module. Currently, the **B_KEEP_LOADED** flag is the only one available; as expected, it tells the kernel not to unload your module when nobody is using it; normally, the first time your module is requested by someone calling **get_module()**, the kernel loads it. With each subsequent call to **get_module()**, a reference count is incremented. Every time **put_module()** is called to release the module, the reference count is decremented. When the counter reaches zero, the module is unloaded. **B_KEEP_LOADED** prevents unloading from taking place.

std_ops is a pointer to a function that your module must provide. This function is called to handle standard module operations. Currently, there are only two of these operations (initialization and uninitialization). Your module's **std_ops()** function will probably look something like this:

```
static status_t std_ops(int32 op, ...) {
   switch(op) {
      case B_MODULE_INIT:
         /* do whatever you need to do */
         break;
      case B_MODULE_UNINIT:
         /* do whatever you need to do */
         break;
      default:
         return B_ERROR;   /* necessary, for future expansion */
   }
   return B_OK;
}
```

It's important to return **B_ERROR** for any unknown operations, in case future versions of the kernel define additional operations.

Exporting your module to the outside world is similar to publishing device driver hooks, but since you define the hooks yourself, it's slightly more involved. Your module needs to have a filled−out version of your module's information structure, like this:

```
static struct my_module_info my_module {
   {
      MY_MODULE_NAME,   /* module name */
      0,   /* flags */
      std_ops
   },
   function1,
   function2,
   configure
};
```

When loading your module, the kernel looks for a symbol called "modules" that contains a list of pointers to the modules you export, terminated by a **NULL**:

```
_EXPORT module_info *modules[] = {
   (module_info *) &my_module,
   NULL
};
```

This is how the kernel finds out what modules are available for use by drivers (or by other modules). See the "Using Modules" section for details on how modules are accessed by other drivers or modules.

Using Modules

Modules provide a means for multiple drivers to share common functionality; for example, if a variety of types of device might be accessed on the same bus, a module might be created to provide a common interface to the bus.

Your driver can access these modules via the kernel functions **`get_module()`** and **`put_module()`**, which obtain and release references to a specified module. When you call **`get_module()`**, you obtain a structure that provides information about the module, plus pointers to the module's functions. The module is defined in a header file provided by the module's author, similar to this:

```
#define MY_MODULE_NAME "generic/mymodule/v1"

struct my_module_info {
   module_info module;
   int32 (*function1)();
   int32 (*function2)();
   void (*configure)(int32 parameter, int32 value);
};
```

When you want to access the module's functions, you call **`get_module()`** to get a pointer to this structure from the kernel:

```
struct my_module_info *minfo = NULL;

/* get a pointer to the module */

get_module(MY_MODULE_NAME, (module_info **) &minfo);
```

Once you've done this, you can call the module's functions through the structure:

```
minfo->configure(0, 10);
```

When you're done with the module, you should call **`put_module()`** to release it. The kernel loads and unloads modules as needed, and properly calling **`put_module()`** lets the kernel do its job.

```
put_module(MY_MODULE_NAME);
```

If you want a better understanding of how modules work, see the "Writing Modules" section.

The area_malloc Module

Declared in: drivers/area_malloc.h

The area_malloc module provides a means for your driver to allocate memory in areas instead of on the heap. It provides **malloc()**, **calloc()**, **realloc()**, and **free()** functions that work just like their POSIX counterparts, except they require a pool argument as their first input.

These functions aren't safe to call from interrupt handlers; they may block on semaphores.

The area_malloc functions are thread−safe in relation to one another, but not in relation to **delete_pool()**. Be sure you don't call **delete_pool()** on the pool you're using until you know none of the other functions might be called. **create_pool()** and **delete_pool()** are safe in relation to each other.

When the last user of the module puts it away, any remaining pools are automatically deleted.

Module Functions

create_pool() , delete_pool()

```
const void *create_pool(uint32 addressSpec, size_t size,
    uint32 lockSpec, uint32 protection)

status_t delete_pool(const void *poolID)
```

create_pool() creates a new pool of memory from which to allocate. The parameters are the same as those used by **create_area()**, so you have complete control over the area's characteristics (except for its name). Returns an opaque pool idenfityer, or **NULL** if the creation failed. The ability to share resources allocated from the pool is determined by the permissions and protections used to create the area.

delete_pool() deletes the pool specified by the opaque *poolID* given. Any pointers returned by the other functions in the module are immediately invalid. Returns **B_OK** if the pool was deleted, otherwise **B_ERROR**.

See also: **create_area()** in the Kernel Kit.

malloc() , calloc() , realloc()

```
void *malloc(const void *poolID, size_t size)

void *calloc(const void *poolID, size_t numMembers, size_t size)

void *realloc(const void *poolID, void *ptr, size_t size)
```

malloc() allocates a block of *size* bytes and returns a pointer to it.

calloc() allocates a block that can contain *numMembers* items of the specified *size* and returns a poiner to it.

realloc() resizes the memory block pointed to by *ptr* to the indicated *size*. Resizing a block can require that the memory be relocated, so this function returns the new pointer.

Each of these operations functions in the pool specified by *poolID*.

If there's not enough memory to allocate the requested block, these functions return **NULL**.

free()

```
void free(const void *poolID, void *ptr)
```

Releases the memory block pointed to by *ptr* from the pool specified by *poolID*.

Constants

B_AREA_MALLOC_MODULE_NAME

Declared in: <drivers/area_malloc.h>

The **B_AREA_MALLOC_MODULE_NAME** constant identifies the area_malloc module; use this constant to open the module.

Driver Settings API

Declared in: drivers/driver_settings.h

If your driver is loaded before the file system for the disk on which your settings file resides, your driver might not be able to load its settings using Posix calls. Also, a robust method for reading settings fileseven if they might have become corruptedcan help the system be more stable; if your driver crashes trying to read its settings, the entire system is in jeopardy.

The driver settings API provides easy, safe access to boolean and string settings, and is available to all drivers and modules. If your driver has more complex settings, the **`get_driver_settings()`** function is available to retrieve all your settings in a hierarchical tree.

The boot loader reads the settings files from the boot volume and passes them to the kernel for distribution to the drivers upon request. The boot loader also lets the user add to these settings at boot time; a line of the form "filename:parameters" in the advanced safe mode menu will add "parameters" to the end of the specified settings file. This can be used to change debugging information and to test different options while developing your driver.

Using the Driver Settings API

Using the API is very simple. Just follow these basic steps:

- Call **`load_driver_settings()`** to load the settings data.
- Use **`get_driver_settings()`** or **`get_driver_parameter()`** and **`get_driver_boolean_parameter()`** to read the settings.
- Call **`unload_driver_settings()`** when you're done.

The Settings File

Driver settings files are kept in **~/config/settings/kernel/drivers**.

The settings file is formatted like this:

- Words beginning with "#" indicate that the rest of the line should be treated as a comment.
- Parameters can have values and subparameters. A parameter has the following form in the settings file:

  name [value]* [{
  [parameter]*
  }] ['n',',']

  Where [...] indicates an optional part, and [...]* indicates an optional repeated part.

- Names and values may not contain spaces unless the spaces are preceded by a backslash (') or the words are enclosed in quotes.

Here's an example settings file:

```
device 0 {
  attribute1 value
  attribute2 value
}
device 1 {
  attribute1 value
}
```

For this settings file, **`get_driver_settings()`** will return a pointer to the following tree:

```
driver_settings = {
  parameter_count = 2
  parameters = {
    name = "device"
    value_count = 1
    values = { "0" }
    parameter_count = 2
    parameters = {
      name = "attributes1"
      value_count = 1
      values = "value"
      parameter_count = 0
      parameters = NULL
    },
    {
      name = "attribute2"
      value_count = 1
      values = "value"
      parameter_count = 0
      parameters = NULL
    }
  },
  {
    name = "device"
    value_count = 1
    values = { "1" }
    parameter_count = 1
```

```
    parameters = {
      name = "attribute1"
      value_count = 1
      values = "value"
      parameter_count = 0
      parameters = NULL
    }
  }
}
```

Loading the Settings

To load the driver's settings, you need to call **`load_driver_settings()`**. For example, if your driver's name is "xr_joystick", you might do this:

```
void *handle = load_driver_settings("xr_joystick");
```

The handle is then used when calling the other driver settings functions, to indicate which driver's settings you want to reference. This opaque reference protects you against any future changes in the kernel.

Reading the Settings

There are three functions you can use to read driver settings:

- **`get_driver_boolean_parameter()`** returns a boolean parameter's value.
- **`get_driver_parameter()`** returns a string parameter's value.
- **`get_driver_settings()`** returns all the settings at once, encapsulated in a hierarchical format.

Reading a Boolean Parameter

Let's look at a simple driver that has one boolean parameter, "debug", that enables a special debug mode. The value of this parameter is represented in the settings file by a line "debug value", where value is either "true" or "false". By default, if there's no setting for the debug parameter, false should be assumed. If the parameter is specified but no value is included, we want to assume that the user means true.

Our code to read this setting looks like this:

```
void *handle = load_driver_settings("xr_joystick");
bool debug = get_driver_boolean_parameter(handle, "debug", false,
           true);
unload_driver_settings(handle);
```

If there's no settings file, **`load_driver_settings()`** will return **`NULL`**. In this case, **`get_driver_boolean_parameter()`** will return **`false`** (the value we're passing as the *unknownValue* argument).

If there's a settings file, but the debug entry isn't found, the *unknownValue* argument is returned. Even though the handle is valid, the function can't find a value for that argument, so it uses this as the default.

If the file contains a line starting with "debug", the second word on the line is used as the value. If no value is specified, **`true`** is returned (the value of the *noArgValue* argument to **`get_driver_boolean_parameter()`**). Otherwise the following is done:

- If the value is "1", "true", "yes", "on", "enable", or "enabled", true is returned.
- If the value is "0", "false", "no", "off", "disable", or "disabled", false is returned.
- If the value matches none of these strings, it's treated as if no entry were found, and *unknownValue* is returned.

If more than one line containing the word "debug" is found, the last one in the file is used. This lets the user override, at boot time, the value previously specified in the settings file.

Reading a String Parameter

Reading string parameters works in much the same way, using the **`get_driver_parameter()`** function. The only difference is that the string returned will be **`NULL`** if the parameter is missing, or the file doesn't exist.

Reading All Parameters

If your driver has more complex parameters (such as parameters with multiple values, or with subparameters), you can read the entire settings tree using the **`get_driver_settings()`** function.

The **`driver_settings`** structure contains the root of the settings tree:

```
typedef struct driver_settings {
   int parameter_count;
   struct driver_parameter *parameters;
};
```

Each parameter is described by the driver_parameter structure:

```
typedef struct driver_parameter {
   char *name;
   int value_count;
   char **values;
   int parameter_count;
   struct driver_parameter *parameters;
};
```

C Functions

get_driver_boolean_parameter()

```
bool get_driver_boolean_parameter(void *handle,
    const char *keyName,
    bool unknownValue,
    bool noArgValue)
```

Returns the value of a given boolean parameter. The driver settings file is specified by the *handle*, as returned by **load_driver_settings()**. The parameter's name is given by *keyName*. If the parameter isn't found, *unknownValue* is returned. If the parameter exists but has no value, *noArgValue* is returned. This lets you easily deal with these two conditions, providing appropriate default values without additional code to check for error conditions.

If the handle is **NULL**, *unknownValue* is returned.

get_driver_parameter()

```
const char *get_driver_parameter(void *handle,
    const char *keyName,
    const char *unknownValue,
    const char *noArgValue)
```

Returns the value of a given string parameter. The driver settings file is specified by the *handle*, as returned by **load_driver_settings()**. The parameter's name is given by *keyName*. If the parameter isn't found, *unknownValue* is returned. If the parameter exists but has no value, *noArgValue* is returned. This lets you easily deal with these two conditions, providing appropriate default values without additional code to check for error conditions.

The special keyName value **B_SAFEMODE_SAFE_MODE** can be used if you want to find out whether or not BeOS was booted in safe mode; the value will be true if BeOS is running in safe mode, or false if a normal boot was performed.

If the handle is **NULL**, *unknownValue* is returned.

get_driver_settings()

```
const driver_settings *get_driver_settings(void *handle)
```

Returns the values of all parameters in encapsulated form.

load_driver_settings() , unload_driver_settings()

```
void *load_driver_settings(const char *driverName)

status_t unload_driver_settings(void *handle)
```

load_driver_settings() loads the settings for the driver specified by *driverName*, and returns a handle that should be used for calls to other driver settings functions. If you want to access the safe mode settings, pass **B_SAFEMODE_DRIVER_SETTINGS** Returns **NULL** if no settings are available for the driver.

unload_driver_settings() unloads the settings for the driver whose settings file is specified by *handle*. You should always call this function when you're done reading the settings.

Defined Types

driver_parameter

```
typedef struct driver_parameter {
        char *name;
        int value_count;
        char **values;
        int parameter_count;
        struct driver_parameter *parameters;
    };
```

Describes a subtree of parameters.

---

driver_settings

```
typedef struct driver_settings {
        int parameter_count;
        struct driver_parameter *parameters;
    };
```

Encapsulates all the settings for a driver.

Constants and Defined Types

This section covers constants and types defined for use by kernel drivers and modules.

Constants

Current Driver API Version

Declared in: <drivers/Drivers.h>

The **B_CUR_DRIVER_API_VERSION** constant indicates what version of the driver API your driver will be built to.

See also: "Symbols Drivers Export"

Driver Control Opcodes

Declared in: <drivers/Drivers.h>

B_GET_DEVICE_SIZE	Returns a size_t indicating the device size in bytes.
B_SET_DEVICE_SIZE	Sets the device size to the value pointed to by *data*.
B_SET_NONBLOCKING_IO	Sets the device to use nonblocking I/O.
B_SET_BLOCKING_IO	Sets the device to use blocking I/O.
B_GET_READ_STATUS	Returns `true` if the device can read without blocking, otherwise `false` .
B_GET_WRITE_STATUS	Returns `true` if the device can write without blocking, otherwise `false` .
B_GET_GEOMETRY	Fills out the specified device_geometry structure to describe the device.
B_GET_DRIVER_FOR_DEVICE	Returns the path of the driver executable handling the device.
B_GET_PARTITION_INFO	Returns a partition_info structure for the device.
B_SET_PARTITION	Creates a user−defined partition. *data* points to a partition_info structure.
B_FORMAT_DEVICE	Formats the device. *data* should point to a boolean value. If this is `true` , the device is formatted low−level. If it's `false` , <<<unclear>>>
B_EJECT_DEVICE	Ejects the device.
B_GET_ICON	Fills out the specified device_icon structure to describe the device's icon.
B_GET_BIOS_GEOMETRY	Fills out a device_geometry structure to describe the device as the BIOS sees it.
B_GET_MEDIA_STATUS	Gets the status of the media in the device by placing a status_t at the location pointed to by *data*.
B_LOAD_MEDIA	Loads the media, if this is supported. <<<what does that mean?>>>
B_GET_BIOS_DRIVE_ID	Returns the BIOS ID for the device.
B_SET_UNINTERRUPTABLE_IO	Prevents control−C from interrupting I/O.
B_SET_INTERRUPTABLE_IO	Allows control−C to interrupt I/O.
B_FLUSH_DRIVE_CACHE	Flushes the drive's cache.
B_GET_NEXT_OPEN_DEVICE	Iterates through open devices; data points to an `open_device_iterator` .

B_ADD_FIXED_DRIVER	For internal use only.
B_REMOVE_FIXED_DRIVER	For internal use only.
B_AUDIO_DRIVER_BASE	Base for codes in audio_driver.h.
B_MIDI_DRIVER_BASE	Base for codes in midi_driver.h.
B_JOYSTICK_DRIVER_BASE	Base for codes in joystick.h.
B_GRAPHIC_DRIVER_BASE	Base for codes in graphic_driver.h.
B_DEVICE_OP_CODES_END	End of Be-defined control IDs.

B_GET_MEDIA_STATUS can return the following values:

Defined Types

device_geometry

Declared in: <drivers/Drivers.h>

```
typedef struct {
    uint32 bytes_per_sector ;
    uint32 sectors_per_track ;
    uint32 cylinder_count ;
    uint32 head_count ;
    uchar device_type ;
    bool removable ;
    bool read_only ;
    bool write_once ;
} device_geometry
```

The device_geometry structure is returned by the **B_GET_GEOMETRY** driver control function. Its fields are:

- *bytes_per_sector* indicates how many bytes each sector of the disk contains.
- *sectors_per_track* indicates how many sectors each disk track contains.
- *cylinder_count* indicates the number of cylinders the disk contains.
- *head_count* indicates how many heads the disk has.
- *device_type* specifies the type of device; there's a list of device type definitions below.
- *removable* is **true** if the device's media can be removed from the drive, and is **false** otherwise.
- *read_only* is **true** if the media is read-only (such as CD-ROM), or **false** if the media can be both read from and written .
- *write_once* is **true** if the media can only be written to once (such as CD-recordable), or **false** if there's no limit to the number of times the media can be written to.

If you need to compute the total size of the device in bytes, you can obtain this figure using the following simple formula:

```
disk_size = geometry.cylinder_count * geometry.sectors_per_track *
        geometry.head_count * geometry.bytes_per_sector;
```

The device type returned in *device_type* is:

B_DISK	Hard disk, floppy disk, etc.
B_TAPE	Tape drive
B_PRINTER	Printer
B_CPU	CPU device

B_WORM	Write−once, read−many device (like CD−recordable)
B_CD	CD−ROM
B_SCANNER	Scanner
B_OPTICAL	Optical device
B_JUKEBOX	Jukebox device
B_NETWORK	Network device

device_hooks

Declared in: <drivers/Drivers.h>

```
typedef struct {
    device_open_hook open ;
    device_close_hook close ;
    device_free_hook free ;
    device_control_hook control ;
    device_read_hook read ;
    device_write_hook write ;
    device_select_hook select ;
    device_deselect_hook deselect ;
    device_readv_hook readv ;
    device_writev_hook writev ;
} device_hooks
```

This structure is used by device drivers to export their function hooks to the kernel.

device_icon

Declared in: <drivers/Drivers.h>

```
typedef struct {
    int32 icon_size ;
    void * icon_data ;
} device_icon
```

When you want to obtain an icon for a specific device, call **ioctl()** on the open device, specifying the **B_GET_ICON** opcode. Pass in data a pointer to a device_icon structure in which *icon_size* indicates the size of icon you want and *icon_data* points to a buffer large enough to receive the icon's data.

icon_size can be either B_MINI_ICON, in which case the buffer pointed to by icon_data should be large enough to receive a 16x16 8−bit bitmap (256−byte), or B_LARGE_ICON, in which case the buffer should be large enough to receive a 32x32 8−bit bitmap (1024−byte). The most obvious way to set up this buffer would be to create a BBitmap of the appropriate size and color depth and use its buffer, like this:

```
BBitmap bits(BRect(0, 0, B_MINI_ICON−1, B_MINI_ICON−1, 0, B_CMAP8));
device_icon iconrec;

iconrec.icon_size = B_MINI_ICON;
iconrec.icon_data = bits.Bits();
status_t err = ioctl(dev_fd, B_GET_ICON, &iconrec);
if (err == B_OK) {
   /* enjoy the icon */
   ...
   view−>DrawBitmap(bits);
} else {
   /* I don't like icons anyway */
   }
```

driver_path

Declared in: <drivers/Drivers.h>

```
typedef char driver_path [256];
```

Used by the **B_GET_DRIVER_FOR_DEVICE** control function to return the pathname of the specified device.

---

open_device_iterator

Declared in: <drivers/Drivers.h>

```
typedef struct {
        uint32 cookie ;
        char device [256];
    } open_device_iterator
```

Used by the **B_GET_NEXT_OPEN_DEVICE** control function. The first time you call this function, your open_device_iterator should have *cookie* initialized to 0. Then just keep calling it over and over; each time you'll get the name of the next open device. When an error is returned, you're done.

---

partition_info

Declared in: <drivers/Drivers.h>

```
typedef struct {
        off_t offset ;
        off_t size ;
        int32 logical_block_size ;
        int32 session ;
        int32 partition ;
        char device [256];
    } partition_info
```

The partition_info structure describes a disk partition, and is used by the **B_GET_PARTITION_INFO** and **B_SET_PARTITION** control commands.

The fields are:

- *offset* is the offset, in bytes, from the beginning of the disk to the beginning of the partition.
- *size* is the size, in bytes, of the partition.
- *logical_block_size* is the block size with which the file system was written to the partition.
- *session* and *partition* are the session and partition ID numbers for the partition.
- *device* is the pathname of the physical device on which the partition is located.

Functions for Drivers & Modules

The kernel exports a number of functions that device drivers can call. The device driver accesses these functions directly in the kernel, not through a library.

Remember when writing a driver that calls one of these functions to link against **_KERNEL_**. This will instruct the loader to dynamically locate the symbols in the current kernel when the driver is loaded.

---

acquire_spinlock() , release_spinlock() , spinlock

Declared in: be/drivers/KernelExport.h

void **acquire_spinlock(**spinlock **lock***)**

void **release_spinlock(**spinlock **lock***)**

typedef vlong **spinlock**

Spinlocks are mutually exclusive locks that are used to protect sections of code that must execute atomically. Unlike semaphores, spinlocks can be safely used when interrupts are disabled (in fact, you *must* have interrupts disabled).

To create a spinlock, simply declare a **spinlock** variable and initialize it 0:

```
spinlock lock = 0;
```

The functions acquire and release the *lock* spinlock. When you acquire and release a spinlock, you *must* have interrupts disabled; the structure of your code will look like this:

```
cpu_status former = disable_interrupts();
acquire_spinlock(&lock);
/* critical section goes here */
release_spinlock(&lock);
restore_interrupts(former);
```

The spinlock should be held as briefly as possible, and acquisition must not be nested within the critical section.

Spinlocks are designed for use in a multi−processor system (on a single processor system simply turning off interrupts is enough to guarantee that the critical section will be atomic). Nonetheless, you *can* use spinlocks on a single processoryou don't have to predicate your code based on the number of CPUs in the system.

---

add_debugger_command() see kernel_debugger()

---

add_timer() , cancel_timer() , timer_hook , qent , timer

Declared in: be/drivers/KernelExport.h

typedef int32 (***timer_hook**)(timer *)

struct **qent** = {
    int64 key;
    qent *next;
    qent *prev;
    }

struct **timer** = {
    qent entry;
    uint16 flags;
    uint16 cpu;
    timer_hook hook;
    bigtime_t period;
    }

status_t **add_timer(**timer ***theTimer*, timer_hook *hookFunction*, bigtime_t *period*, int32 *flags***)**

bool **cancel_timer(**timer_t ***theTimer***)**

add_timer() installs a new timer interrupt. A timer interrupt causes the specified *hookFunction* to be called when the desired amount of time has passed. On entry, you should pass a pointer to a timer structure in *theTimer*; this will be filled out with data describing the new timer interrupt you've installed. The *flags* argument provides control over how the timer functions, which affects the meaning of the *period* argument as follows:

B_ONE_SHOT_ABSOLUTE_TIMER	The timer will fire once at the system time specified by *period*.
B_ONE_SHOT_RELATIVE_TIMER	The timer will fire once in approximately *period* microseconds.
B_PERIODIC_TIMER	The timer will fire every *period* microseconds, starting in *period* microseconds.

cancel_timer() cancels the specified timer. If it's already fired, it returns true; otherwise false is returned. It's guaranteed that once **cancel_timer()** returns, if the timer was in the process of running when **cancel_timer()** was called, the timer function will be finished executing. The only exception to this is if **cancel_timer()** was called from inside a timer handler (in which case trying to wait for the handler to finish running would result in deadlock).

RETURN CODES

B_OK. The timer was installed (**add_timer()** only).

- **B_BAD_VALUE**. The timer couldn't be installed because the period was invalid (probably because a relative time or period was negative; unfortunately, Be hasn't mastered the intricacies of installing timers to fire in the past).

---

call_all_cpus()

Declared in: be/drivers/KernelExport.h

void **call_all_cpus(**void (**func*)(void *, int), void **cookie***)*

Calls the function specified by *func* on all CPUs. The *cookie* can be anything your needs require.

---

cancel_timer() see add_timer()

---

disable_interrupts() , restore_interrupts() , cpu_status

Declared in: be/drivers/KernelExport.h

typedef ulong **cpu_status**

cpu_status **disable_interrupts(**void**)**

void **restore_interrupts(**cpu_status *status***)**

These functions disable and restore interrupts on the CPU that the caller is currently running on. **disable_interrupts()** returns its previous state (i.e. whether or not interrupts were already disabled). **restore_interrupts()** restores the previous *status* of the CPU, which should be the value that **disable_interrupts()** returned:

```
cpu_status former = disable_interrupts();
...
restore_interrupts(former);
```

As long as the CPU state is properly restored (as shown here), the disable/restore functions can be nested.

See also: **install_io_interrupt_handler()**

---

dprintf() , set_dprintf_enabled() , panic()

Declared in: be/drivers/KernelExport.h

void **dprintf(**const char **format*, ...**)**

bool **set_dprintf_enabled(**bool *enabled***)**

void **panic(**const char **format*, ...**)**

dprintf() is a debugging function that has the same syntax and behavior as standard C **printf()**, except that it writes its output to the serial port at a data rate of 19,200 bits per second. The output is sent to **/dev/ports/serial4** on BeBoxes, **/dev/modem** on Macs, and **/dev/ports/serial1** on Intel

machines. By default, **dprintf()** is disabled.

set_dprintf_enabled() enables **dprintf()** if the *enabled* flag is **true**, and disables it if the flag is **false**. It returns the previous enabled state, thus permitting intelligent nesting:

```
/* Turn on dprintf */
bool former = set_dprintf_enabled(true);
...
/* Now restore it to its previous state. */
set_dprintf_enabled(former);
```

panic() is similar to **dprintf()**, except it hangs the computer after printing the message.

---

get_memory_map() , physical_entry

Declared in: be/drivers/KernelExport.h

long **get_memory_map(** const void *address, ulong *numBytes*,
physical_entry *table, long *numEntries* **)**

typedef struct { void *address;
ulong size;
} **physical_entry**

Returns the physical memory chunks that map to the virtual memory that starts at *address* and extends for *numBytes*. Each chunk of physical memory is returned as a **physical_entry** structure; the series of structures is returned in the *table* array. (which you have to allocate yourself). *numEntries* is the number of elements in the array that you're passing in. As shown in the example, you should lock the memory that you're about to inspect:

```
physical_entry table[count];
lock_memory(addr, extent, 0);
get_memory_map(addr, extent, table, count);
. . .
unlock_memory(someAddress, someNumberOfBytes, 0);
```

The end of the *table* array is indicated by (**size** == 0):

```
long k;
while (table[k].size > 0) {
   /* A legitimate entry */
   if (++k == count) {
      /* Not enough entries */
      break; }
}
```

If all of the entries have non−zero sizes, then table wasn't big enough; call **get_memory_map()** again with more table entries.

RETURN CODES

The function always returns **B_OK**.

See also: **lock_memory()**, **start_isa_dma()**

---

has_signals_pending()

Declared in: be/drivers/KernelExport.h

int **has_signals_pending(** struct thread_rec *thr **)**

Returns a bitmask of the currently pending signals for the current thread. *thr* should always be **NULL**; passing other values will yield meaningless results. **has_signals_pending()** returns 0 if no signals are pending.

---

install_io_interrupt_handler() , remove_io_interrupt_handler()

Declared in: be/drivers/KernelExport.h

long **install_io_interrupt_handler(** long *interrupt_number*,
interrupt_handler *handler*,
void *data, ulong *flags* **)**

long **remove_io_interrupt_handler(** long *interrupt_number*,

```
    interrupt_handler handler,
    void *data)
```

install_io_interrupt_handler() adds the handler *function* to the chain of functions that will be called each time the specified *interrupt* occurs. This function should have the following syntax:

```
int32 handler(void *data)
```

The *data* passed to **install_io_interrupt_handler()** will be passed to the handler function each time it's called. It can be anything that might be of use to the handler, or **NULL**. If the interrupt handler must return one of the following values:

B_UNHANDLED_INTERRUPT	The interrupt handler didn't handle the interrupt; the kernel will keep looking for someone to handle it.
B_HANDLED_INTERRUPT	The interrupt handler handled the interrupt. The kernel won't keep looking for a handler to handle it.
B_INVOKE_SCHEDULER	The interrupt handler handled the interrupt. This tells the kernel to invoke the scheduler immediately after the handler returns.

If **B_INVOKE_SCHEDULER** is returned by the interrupt handler, the kernel will immediately invoke the scheduler, to dispatch processor time to tasks that need handling. This is especially useful if your interrupt handler has released a semaphore (see **release_sem_etc()** in the Kernel Kit).

The *flags* parameter is a bitmask of options. The only option currently defined is **B_NO_ENABLE_COUNTER**. By default, the OS keeps track of the number of functions handling a given interrupt. If this counter changes from 0 to 1, then the system enables the irq for that interrupt. Conversely, if the counter changes from 1 to 0, the system disables the irq. Setting the **B_NO_ENABLE_COUNTER** flag instructs the OS to ignore the handler for the purpose of enabling and disabling the irq.

install_io_interrupt_handler() returns **B_OK** if successful in installing the handler, and **B_ERROR** if not. An error occurs when either the *interrupt_number* is out of range or there is not enough room left in the interrupt chain to add the handler.

remove_io_interrupt() removes the named *interrupt* from the interrupt chain. It returns **B_OK** if successful in removing the handler, and **B_ERROR** if not.

io_card_version() see motherboard_version()

kernel_debugger() , add_debugger_command() , remove_debugger_command() , load_driver_symbols() , kprintf() , parse_expression()

Declared in: be/drivers/KernelExport.h

```
void kernel_debugger(const char *string)

int add_debugger_command(char *name, int (*func)(int, char **), char *help)

int remove_debugger_command(char *name, int (*func)(int, char **))

int load_driver_symbols(const char *driverName)

void kprintf(const char *format, ...)

ulong parse_expression(const char *string)
```

kernel_debugger() drops the calling thread into a debugger that writes its output to the serial port at 19,200 bits per second, just as **dprintf()** does. This debugger produces *string* as its first message; it's not affected by **set_dprintf_enabled()**.

kernel_debugger() is identical to the **debugger()** function documented in the Kernel Kit, except that it works in the kernel and engages a different debugger. Drivers should use it instead of **debugger()**.

add_debugger_command() registers a new command with the kernel debugger. When the user types in the command *name*, the kernel debugger calls func with the remainder of the command line as *argc*/*argv*−style arguments. The help string for the command is set to *help*.

remove_debugger_command() removes the specified kernel debugger command.

load_driver_symbols() loads symbols from the specified kernel driver into the kernel debugger. **driver_name** is the path−less name of the driver which must be located in one of the standard kernel driver directories. The function returns **B_OK** on success and **B_ERROR** on failure.

kprintf() outputs messages to the serial port. It should be used instead of **dprintf()** from new debugger commands because

dprintf() depends too much upon the state of the kernel to be reliable from within the debugger.

parse_expression() takes a C expression and returns the result. It only handles integer arithmetic. The logical and relational operations are accepted. It can also supports variables and assignments. This is useful for strings with multiple expressions, which should be separated with semicolons. Finally, the special variable "." refers to the value from the previous expression. This function is designed to help implement new debugger commands.

See also: **debugger()** in the Kernel Kit

---

kprintf() see kernel_debugger()

---

load_driver_symbols() see kernel_debugger()

---

lock_memory() , unlock_memory()

Declared in: be/drivers/KernelExport.h

long **lock_memory(**void *_address_, ulong _numBytes_, ulong _flags_**)**

long **unlock_memory(**const void *_address_, ulong _numBytes_, ulong _flags_**)**

lock_memory() makes sure that all the memory beginning at the specified virtual *address* and extending for *numBytes* is resident in RAM, and locks it so that it won't be paged out until **unlock_memory()** is called. It pages in any of the memory that isn't resident at the time it's called. It is typically used in preparation for a DMA transaction.

The *flags* field contains a bitmask of options. Currently, two options, **B_DMA_IO** and **B_READ_DEVICE**, are defined. **B_DMA_IO** should be set if any part of the memory range will be modified by something other than the CPU while it's locked, since that change won't otherwise be noticed by the system and the modified pages may not be written to disk by the virtual memory system. Typically, this sort of change is performed through DMA. **B_READ_DEVICE**, if set, indicates that the caller intends to fill the memory (read *from* the device). If cleared, it indicates the memory will be written to the device and will not be altered.

unlock_memory() releases locked memory and should be called with the same flags as passed into the corresponding **lock_memory()** call.

Each of these functions returns **B_OK** if successful and **B_ERROR** if not. The main reason that **lock_memory()** would fail is that you're attempting to lock more memory than can be paged in.

---

map_physical_memory()

Declared in: be/drivers/KernelExport.h

area_id **map_physical_memory(**const char *_areaName_, void *_physicalAddress_,
size_t _numBytes_, uint32 _spec_, uint32 _protection_,
void **_virtualAddress_**)**

This function allows you to map the memory in physical memory starting at *physicalAddress* and extending for *numBytes* bytes into your team's address space. The kernel creates an area named *areaName* mapped into the memory address *virtualAddress* and returns its area_id to the caller. *numBytes* must be a multiple of **B_PAGE_SIZE** (4096).

spec must be either **B_ANY_KERNEL_ADDRESS** or **B_ANY_KERNEL_BLOCK_ADDRESS**. If *spec* is **B_ANY_KERNEL_ADDRESS**, the memory will begin at an arbitrary location in the kernel address space. If *spec* is **B_ANY_KERNEL_BLOCK_ADDRESS**, then the memory will be mapped into a memory location aligned on a multiple of **B_PAGE_SIZE**.

protection is a bitmask consisting of the fields **B_READ_AREA** and **B_WRITE_AREA**, as discussed in **create_area()**.

create_area() returns an area_id for the newly−created memory if successful or an error code on failure. The error codes are the same as those for **create_area()**.

See also: **create_area()**

---

motherboard_version() , io_card_version()

Declared in: be/drivers/KernelExport.h

long **motherboard_version(**void**)**

long **io_card_version(**void**)**

These functions return the current versions of the motherboard and of the I/O card. These functions are only available on PowerPC−based systems (they're intended for use on the BeBox).

panic() see dprintf()

parse_expression() see kernel_debugger()

platform()

Declared in: be/drivers/KernelExport.h

platform_type **platform(**void**)**

Returns the current platform, as defined in **<kernel/OS.h>**.

register_kernel_daemon() , unregister_kernel_daemon()

Declared in: be/drivers/KernelExport.h

int **register_kernel_daemon(**void (**func*)(void *, int), void **arg*, int *freq***)**

int **unregister_kernel_daemon(**void (**func*)(void *, int), void **arg***)**

Adds or removes daemons from the kernel. A kernel daemon function is executed approximately once every *freq*/10 seconds. The kernel calls *func* with the arguments *arg* and an iteration value that increases by *freq* on successive calls to the daemon function.

release_spinlock() see acquire_spinlock()

remove_io_interrupt_handler() see install_io_interrupt_handler()

restore_interrupts() see disable_interrupts()

set_dprintf_enabled() see dprintf()

send_signal_etc()

Declared in: be/drivers/KernelExport.h

int **send_signal_etc(**pid_t *thid*, uint *sig*, uint32 *flags***)**

This function is a counterpart to **send_signal()** in the Posix layer, which is not exported for drivers.

thid is the thread_id of the thread the signal should be sent to, and sig is the signal type to send, just like in **send_signal()**. The *flags* argument can be used to specify flags to control the function:

B_CHECK_PERMISSION	The signal will only be sent if the destination thread's uid and euid are the same as the caller's.
B_DO_NOT_RESCHEDULE	The kernel won't call the scheduler after sending the signal. You should specify this flag when calling **send_signal_etc()** from an interrupt handler.

RETURN CODES

B_OK. The signal was sent.

- **B_BAD_VALUE**. The signal type is invalid.
- **B_BAD_THREAD_ID**. The thread ID is invalid.

- **B_NOT_ALLOWED**. The permission check failed (if **B_CHECK_PERMISSION** was specified).

---

spawn_kernel_thread()

Declared in: be/drivers/KernelExport.h

```
thread_id spawn_kernel_thread(thread_entry func, const char *name,
    long priority, void *data)
```

This function is a counterpart to **spawn_thread()** in the Kernel Kit, which is not exported for drivers. It has the same syntax as the Kernel Kit function, but is able to spawn threads in the kernel's memory space.

See also: **spawn_thread()** in the Kernel Kit

---

spin()

Declared in: be/drivers/KernelExport.h

```
void spin(bigtime_t microseconds)
```

Executes a delay loop lasting at least the specified number of *microseconds*. It could last longer, due to rounding errors, interrupts, and context switches.

---

unlock_memory() see lock_memory()

---

unregister_kernel_daemon() see register_kernel_daemon()

Drivers: Master Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

writev_hook()	Writing Drivers
Writing Drivers	Writing Drivers
Writing Drivers	Writing Drivers
Writing Modules	anon
Writing Modules	Writing Modules
Writing Modules	Writing Modules