8 The Kernel Kit

8 The Kernel Kit

The Kernel Kit is a collection of C functions that let you define and control the contexts in which your application operates. There are five main topics in the Kit:

- *Threads and Teams.* A thread is a synchronous computer process. By creating multiple threads, you can make your application perform different tasks at (virtually) the same time. A team is the collection of threads that your application creates.

- *Ports*. A port can be thought of as a mailbox for threads: A thread can write a message to a port, and some other thread (or, less usefully, the same thread) can then retrieve the message.

- *Semaphores*. A semaphore is a system-wide counting variable that can be used as a lock that protects a piece of code. Before a thread is allowed to execute the code, it must acquire the semaphore that guards it. Semaphores can also be used to synchronize the execution of two or more threads.

- *Areas*. The area functions let you allocate large chunks of virtual memory. The two primary features of areas are: They can be locked into the CPU's on-chip memory, and the data they hold can be shared between applications.

- *Images*. An image is compiled code that can be dynamically linked into a running application. By loading and unloading images you can make run-time decisions about the resources that your application has access to. Images are of particular interest to driver designers.

The rest of this chapter describes these topics in detail. The final section, “Miscellaneous Functions, Constants, and Defined Types”, describes the associated API that support the Kit functions.

Threads and Teams

Declared in: <kernel/OS.h>

Overview

A thread is a synchronous computer process that executes a series of program instructions. Every application has at least one thread: When you launch an application, an initial thread—the *main thread*—is automatically created (or *spawned*) and told to run. The main thread executes the ubiquitous **main()** function, winds through the functions that are called from **main()**, and is automatically deleted (or *killed*) when **main()** exits.

The Be operating system is *multi-threaded*: From the main thread you can spawn and run additional threads; from each of these threads you can spawn and run more threads, and so on. All the threads in all applications run concurrently and asynchronously with each other. Furthermore, threads are independent of each other; most notably, a given thread doesn't own the other threads it has spawned. For example, if thread A spawns thread B, and thread A dies (for whatever reason), thread B will continue to run. (But before you get carried away with the idea of leap-frogging threads, you should take note of the caveat in the section "Death and the Main Thread" on page 11,)

Although threads are independent, they do fall into groups called *teams*. A team consists of a main thread and all other threads that "descend" from it (that are spawned by the main thread directly, or by any thread that was spawned by the main thread, and so on). Viewed from a higher level, a team is the group of threads that are created by a single application. You can't "transfer" threads from one team to another. The team is set when the thread is spawned; it remains the same throughout the thread's life.

All the threads in a particular team share the same address space: Global variables that are declared by one thread will be visible to all other threads in that team.

The following sections describe how to spawn, control, and examine threads and teams.

Spawning a Thread

You spawn a thread by calling the **spawn_thread()** function. The function assigns and returns a system-wide **thread_id** number that you use to identify the new thread in subsequent function calls. Valid **thread_id** numbers are positive integers; you can check the success of a spawn thus:

```
thread_id my_thread = spawn_thread(...);

if ((my_thread) < B_NO_ERROR)
    /* failure */
else
    /* success */
```

The arguments to **spawn_thread()**, which are examined throughout this description, supply information such as what the thread is supposed to do, the urgency of its operation, and so on.

Note: A conceptual neighbor of spawning a thread is the act of loading an executable (or loading an *app image*). This is performed by calling the **load_executable()** function. Loading an executable causes a separate program, identified as a file, to be launched by the system. For more information on the **load_executable()** function, see “Images” beginning on page 55.

Telling a Thread to Run

Spawning a thread isn’t enough to make it run. To tell a thread to start running, you must pass its **thread_id** number to either the **resume_thread()** or **wait_for_thread()** function:

- **resume_thread()** starts the new thread running and immediately returns. The new thread runs concurrently and asynchronously with the thread in which **resume_thread()** was called.
- **wait_for_thread()** starts the thread running but doesn’t return until the thread has finished. (You can also call **wait_for_thread()** on a thread that’s already running.)

Of these two functions, **resume_thread()** is the more common means for starting a thread that was created through **spawn_thread()**. **wait_for_thread()**, on the other hand, is often used to start a thread that was created through **load_executable()**.

The Entry Function

When you call **spawn_thread()**, you must identify the new thread’s *entry function*. This is a global C function (or a static C++ member function) that the new thread will execute when it’s told to run. When the entry function exits, the thread is automatically killed by the operating system.

A thread’s entry function assumes the following protocol:

```
long thread_entry(void *data);
```

The protocol signifies that the function can return a value (to whom the value is returned is a topic that will be explored later), and that it accepts a pointer to a buffer of arbitrarily-typed data. (The function’s name isn’t prescribed by the protocol; in other words, an entry function doesn’t *have* to be named “thread_entry”.)

You specify a thread's entry function by passing a pointer to the function as the first argument to **spawn_thread()**; the last argument to **spawn_thread()** is forwarded as the entry function's *data* argument. Since *data* is delivered as a **void ***, you have to cast the value to the appropriate type within your implementation of the entry function. For example, let's say you define an entry function called **lister()** that takes a pointer to a BList object as an argument:

```
long lister(void *data)
{
    /* Cast the argument. */
    BList *listObj = (BList *)data;
    ...
}
```

To create and run a thread that would execute the **lister()** function, you call **spawn_thread()** and **resume_thread()** thus (excluding error checks):

```
BList *listObj = new BList();
thread_id my_thread;

my_thread = spawn_thread(lister, ..., (void *)listObj);
resume_thread(my_thread);
```

The Entry Function's Argument

The **spawn_thread()** function *doesn't* copy the data that *data* points to. It simply passes the pointer through literally. Because of this, you should never pass a pointer that's allocated locally (on the stack).

The reason for this restriction is that there's no guarantee that the entry function will receive *any* CPU attention before the stack frame from which **spawn_thread()** was called is destroyed. Thus, the entry function won't necessarily have a chance to copy the pointed-to data before the pointer vanishes. There are ways around this restriction—for example, you could use a semaphore to ensure that the entry function has copied the data before the calling frame exits. A better solution is to use the **send_data()** function (which *does* copy its data). See "Passing Data to a Thread" on page 12.

Using a C++ Entry Function

If you're up in C++ territory, you'll probably want to define a class member function that you can use as a thread's entry function. Unfortunately, you can't pass a normal (non-static) member function directly as the entry function argument to **spawn_thread()**—the system won't know which object it's supposed to invoke the function on (it won't have a **this** pointer). To get from here to there, you have to declare two member functions:

- a static member function that is, literally, the entry function,
- and a non-static member function that the static function can invoke. This non-static function will perform the intended work of the entry function.

To “connect” the two functions, you pass an object of the appropriate class (through the *data* argument) to the static function, and then allow the static function to invoke the non-static function upon that object. An example is called for: Here we define a class that contains a static function called **entry_func()**, and a non-static function called **entryFunc()**. By convention, these two are private. In addition, the class declares a public **Go()** function, and a private **thread_id** variable:

```
class MyClass : public BObject {
public:
    long Go(void);

private:
    static long entry_func(void *arg);
    long entryFunc(void);
    thread_id my_thread;
};
```

entry_func() is the literal entry function. It doesn’t really do anything—it simply casts its argument as a MyClass object, and then invokes **entryFunc()** on the object:

```
long MyClass::entry_func(void *arg)
{
    MyClass *obj = MyClass *arg;
    return (obj->entryFunc());
}
```

entryFunc() performs the actual work:

```
long MyClass::entryFunc(void)
{
    /* do something here */
    ...
    return (whatever);
}
```

The **Go()** function contains the **spawn_thread()** call that starts the whole thing going:

```
long MyClass::Go(void)
{
    my_thread = spawn_thread(entry_func, ..., this);
    return (resume_thread(my_thread));
}
```

If you aren’t familiar with static member functions, you should consult a qualified C++ textbook. Briefly, the only thing you need to know for the purposes of the technique shown here, is that a static function’s implementation can’t call (non-static) member functions nor can it refer to member data. Maintain the form demonstrated above and you’ll be rewarded in heaven.

Entry Function Return Values

The entry function's protocol declares that the function should return a **long** value when it exits. This value can be captured by sitting in a **wait_for_thread()** call until the entry function exits. **wait_for_thread()** takes two arguments: The **thread_id** of the thread that you're waiting for, and a pointer to a **long** into which the value returned by that thread's entry function will be placed. For example:

```
thread_id other_thread;
long result;

other_thread = spawn_thread(...);
resume_thread(other_thread);

...
wait_for_thread(other_thread, &result);
```

If the target thread is already dead, **wait_for_thread()** returns immediately (with an error code as described in the function's full description), and the second argument will be set to an invalid value. If you're late for the train, you'll miss the boat.

Warning: You *must* pass a valid pointer as the second argument to **wait_for_thread()**; you mustn't pass **NULL** even if you're not interested in the return value.

Thread Names

A thread can be given a name which you assign through the second argument to **spawn_thread()**. The name can be 32 characters long (as represented by the **B_OS_NAME_LENGTH** constant) and needn't be unique—more than one thread can have the same name.

You can look for a thread based on its name by passing the name to the **find_thread()** function; the function returns the **thread_id** of the so-named thread. If two or more threads bear the same name, the **find_thread()** function returns the first of these threads that it finds.

You can retrieve the **thread_id** of the calling thread by passing **NULL** to **find_thread()**:

```
thread_id this_thread = find_thread(NULL);
```

To retrieve a thread's name, you must look in the thread's **thread_info** structure. This structure is described in the **get_thread_info()** function description.

Dissatisfied with a thread's name? Use the **rename_thread()** function to change it. Fool your friends.

Thread Priority

In a multi-threaded environment, the CPUs must divide their attention between the candidate threads, executing a few instructions from this thread, then a few from that thread, and so on. But the division of attention isn't always equal: You can assign a higher or lower *priority* to a thread and so declare it to be more or less important than other threads.

You assign a thread's priority (an integer) as the third argument to **spawn_thread()**. There are two categories of priorities:

- "Time-sharing" priorities (priority values from 1 to 99).
- "Real-time" priorities (100 and greater).

A time-sharing thread (a thread with a time-sharing priority value) is executed only if there are no real-time threads in the ready queue. In the absence of real-time threads, a time-sharing thread is elected to run once every "scheduler quantum" (currently, every three milliseconds). The higher the time-sharing thread's priority value, the greater the chance that it will be the next thread to run.

A real-time thread is executed as soon as it's ready. If more than one real-time thread is ready at the same time, the thread with the highest priority is executed first. The thread is allowed to run without being preempted (except by a real-time thread with a higher priority) until it blocks, snoozes, is suspended, or otherwise gives up its plea for attention.

The Kernel Kit defines seven priority constants. Although you can use other, "in-between" value as the priority argument to **spawn_thread()**, it's strongly suggested that you stick with these:

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORITY	20

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	100
B_URGENT_PRIORITY	110
B_REAL_TIME_PRIORITY	120

Synchronizing Threads

There are times when you may want a particular thread to pause at a designated point until some other (known) thread finishes some task. Here are three ways to effect this sort of synchronization:

- The most general means for synchronizing threads is to use a semaphore. The semaphore mechanism is described in great detail in the major section "Semaphores" beginning on page 31.
- Synchronization is sometimes a side-effect of sending data between threads. This is explained in "Passing Data to a Thread" on page 12, and in the major section "Ports" beginning on page 23
- Finally, you can tell a thread to wait for some other thread to die by calling **wait_for_thread()**, as described earlier.

Controlling a Thread

There are three ways to control a thread while it's running:

- You can put a thread to sleep for some number of microseconds through the **snooze()** function. After the thread has been asleep for the requested time, it automatically resumes execution with its next instruction. **snooze()** only works on the calling thread: The function doesn't let you identify an arbitrary thread as the subject of its operation. In other words, whichever thread calls **snooze()** is the thread that's put to sleep.
- You can suspend the execution of any thread through the **suspend_thread()** function. The function takes a single **thread_id** argument that identifies the thread you wish to suspend. The thread remains suspended until you "unsuspend" it through a call to **resume_thread()** or **wait_for_thread()**.
- You can kill the calling thread through **exit_thread()**. The function takes a single (long) argument that's used as the thread's exit status (to make **wait_for_thread()** happy). More generally, you can kill any thread by passing its **thread_id** to the **kill_thread()** function. **kill_thread()** *doesn't* let you set the exit status.

  Feeling all tense and irritated? Try killing an entire team of threads: The **kill_team()** function is more than a system call. It's therapy.

Death and the Main Thread

As mentioned earlier, the control that's imposed upon a particular thread isn't visited upon the "children" that have been spawned from that thread. (Recall the "thread A spawns thread B then dies" business near the beginning of this overview.) However, the death of an application's main thread can affect the other threads:

> ***When a main thread dies, it takes the team's heap, its statically allocated objects, and other team-wide resources—such as access to standard IO—with it. This may seriously cripple any threads that linger beyond the death of the main thread.***

It's certainly possible to create an application in which the main thread sets up one or more other threads, gets them running, and then dies. But such applications should be rare. In general, you should try to keep your main thread around until all other threads in the team are dead.

Passing Data to a Thread

There are three ways to pass data to a thread:

- Through the argument to the entry function, as described in "The Entry Function's Argument" on page 7.
- By using a port or, at a higher level, by sending a BMessage. Ports are described in the next major section ("Ports"); BMessages are part of the Application Kit.
- By sending data to the thread's message cache through the **send_data()** and **receive_data()** functions, as described below.

The **send_data()** function sends data from one thread to another. With each **send_data()** call, you can send two packets of information:

- a single four-byte value (this is called the *code*),
- and an arbitrarily long buffer of arbitrarily-typed data.

The function's four arguments identify, in order,

- the thread that you want to send the data to,
- the four-byte code,
- a pointer to the buffer of data (a **void ***),
- and the size of the buffer of data, in bytes.

In the following example, the main thread spawns a thread, sends it some data, and then tells the thread to run:

```
main(int argc, char *argv[])
{
    thread_id other_thread;
    long code = 63;
    char *buf = "Hello";

    other_thread = spawn_thread(entry_func, ...);
    send_data(other_thread, code, (void *)buf, strlen(buf));
    resume_thread(other_thread);
    ...
}
```

The **send_data()** call copies the code and the buffer (the second and third arguments) into the target thread's message cache and then (usually) returns immediately. In some cases, the four-byte code is all you need to send; in such cases, the buffer pointer can be **NULL** and the buffer size set to 0.

To retrieve the data that's been sent to it, the target thread (having been told to run) calls **receive_data()**. This function returns the four-byte code directly, and copies the data from the message cache into its second argument. It also returns, by reference in its first argument, the **thread_id** of the thread that sent the data:

```
long entry_func(void *data)
{
    thread_id sender;
    long code;
    char buf[512];

    code = receive_data(&sender, (void *)buf, sizeof(buf));
    ...
}
```

Keep in mind that the message data is *copied* into the second argument; you must allocate adequate storage for the data, and pass, as the final argument to **receive_data()**, the size of the buffer that you allocated. A slightly annoying aspect of this mechanism is that there isn't any way for the data-receiving thread to determine how much data is in the message cache, so it can't know, before it receives the data, what an "adequate" size for its buffer is. If the buffer isn't big enough to accommodate all the data, the left-over portion is simply thrown away. (But at least you don't get a segmentation fault.)

As shown in the example, **send_data()** is called before the target thread is running. This feature of the system is essential in situations where you want the target thread to receive some data as its first act (as demonstrated above). However, **send_data()** isn't limited to this use—you can also send data to a thread that's already running.

Blocking when Sending and Receiving

A thread's message cache isn't a queue; it can only hold one message at a time. If you call **send_data()** twice with the same target thread, the second call will block until the target reads the first transmission through a call to **receive_data().** Analogously, **receive_data()** will block if there isn't (yet) any data to receive.

If you want to make sure that you won't block when receiving data, you should call **has_data()** before calling **receive_data()**. **has_data()** takes a **thread_id** argument, and returns **TRUE** if that thread has a message waiting to be read:

```
if (has_data(find_thread(NULL)))
    code = receive_data(...);
```

You can also use **has_data()** to query the target thread before sending it data. This, you hope, will ensure that the **send_data()** call won't block:

```
if (!has_data(target_thread))
    send_data(target_thread, ...);
```

This usually works, but be aware that there's a race condition between the **has_data()** and **send_data()** calls. If yet another thread sends a message to the same target in that time interval, your **send_data()** (might) block.

Functions

exit_thread(), kill_thread(), kill_team()

void **exit_thread(**long *return_value***)**

long **kill_thread(**thread_id *thread***)**

long **kill_team(**team_id *team***)**

These functions command one or more threads to halt execution:

- **exit_thread()** tells the calling thread to exit with a return value as given by the argument. Declaring the return value is only useful if some other thread is sitting in a **wait_for_thread()** call on this thread.

- **kill_thread()** kills the thread given by the argument. The value that the thread will return to **wait_for_thread()** is undefined and can't be relied upon.

- **kill_team()** kills all the threads within the given team. Again, the threads' return values are random.

Exiting a thread is a fairly safe thing to do—since a thread can only exit itself, it's assumed that the thread knows what it's doing. Killing some other thread or an entire team is a bit more drastic since the death certificate(s) will be delivered at an indeterminate time. Nonetheless, in every case (exiting or killing) the system reclaims the resources that the thread (or team) had claimed. So executing a thread shouldn't cause a memory leak.

Keep in mind that threads die automatically (and their resources are reclaimed) if allowed to exit naturally from their entry functions. You should only need to kill a thread if something has gone screwy.

The kill functions return **B_BAD_THREAD_ID** or **B_BAD_TEAM_ID** if the argument is invalid. Otherwise, they return **B_NO_ERROR**.

find_thread()

thread_id **find_thread(**const char **name***)**

Finds and returns the thread with the given name. A *name* argument of **NULL** returns the calling thread. If *name* doesn't identify a thread, **B_NAME_NOT_FOUND** is returned.

A thread's name is assigned when the thread is spawned. The name can be changed thereafter through the **rename_thread()** function. Keep in mind that thread names needn't be unique: If two (or more) threads boast the same name, a **find_thread()** call on that name returns the first so-named thread that it finds.

get_team_info(), get_nth_team_info()

long **get_team_info(**team_id *team*, team_info **info***)**
long **get_nth_team_info(**long *n*, team_info **info***)**

These functions copy, into the *info* argument, the **team_info** structure for a particular team:

- The **get_team_info()** function retrieves information for the team identified by *team*.
- The **get_nth_team_info()** function retrieves team information for the *n*'th team (zero-based) of all teams currently running on your computer. By calling this function with a monotonically increasing *n* value, you can retrieve information for all teams. When, in this scheme, the function no longer returns **B_NO_ERROR**, all teams will have been visited.

The **team_info** structure is defined as:

```
typedef struct {
        team_id team;
        long thread_count;
        long image_count;
        long area_count;
        thread_id debugger_nub_thread;
        port_id debugger_nub_port;
        long argc;
        char args[64];
} team_info
```

The first field is obvious; the next three reasonably so: They give the number of threads that have been spawned, images that have been loaded, and areas that have been created or cloned within this team.

The debugger fields are used by the, uhm, the...debugger?

The **argc** field is the number of command line arguments that were used to launch the team; **args** is a copy of the first 64 characters from the command line invocation. If this team is an application that was launched through the user interface (by double-clicking, or by accepting a dropped icon), then **argc** is 1 and **args** is the name of the application's executable file.

Both functions return **B_NO_ERROR** upon success. If the designated team isn't found—because *team* in **get_team_info()** isn't valid, or *n* in **get_nth_team_info()** is out-of-bounds—the functions return **BAD_TEAM_ID**.

get_thread_info(), get_nth_thread_info()

long **get_thread_info(**thread_id *thread*, thread_info **info***)**
long **get_nth_thread_info(**team_id *team*, long *n*, thread_info **info***)**

These functions copy, into the *info* argument, the **thread_info** structure for a particular thread:

- The **get_thread_info()** function gets this information for the thread identified by *thread*.
- The **get_nth_thread_info()** function retrieves thread information for the *n*'th thread (zero-based) within the team identified by *team*. If *team* is 0 (zero), all teams are considered. You use this function to retrieve the info structures of all the threads in a team (or in all teams) by repeatedly calling the function with a monotonically increasing value of *n*—the actual value of *n* has no other significance. When, in this scheme, the function no longer returns **B_NO_ERROR**, all candidate threads will have been visited.

The **thread_info** structure is defined as:

```
typedef struct {
        thread_id thread;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        thread_state state;
        long priority;
        sem_id sem;
        double user_time;
        double kernel_time;
        void *stack_base;
        void *stack_end;
} thread_info
```

The fields in the structure are:

- **thread**. The **thread_id** number of the thread.
- **team**. The **team_id** of the thread's team.
- **name**. The name assigned to the thread.
- **state**. What the thread is currently doing (see the thread state constants, below).
- **priority**. The level of attention the thread gets (see the priority constants, below).
- **sem**. If the thread is waiting to acquire a semaphore, this is that semaphore.
- **user_time**. The time, in microseconds, the thread has spent executing user code.
- **kernel_time**. The amount of time the kernel has run on the thread's behalf.
- **stack_base**. A pointer to the first byte in the thread's execution stack.
- **stack_end**. A pointer to the last byte in the thread's execution stack.

The last two fields are only meaningful if you understand the execution stack format. Keep in mind that the stack grows down, from higher to lower addresses. Thus, **stack_base** will always be greater than **stack_end**.

The value of the **state** field is one of following **thread_state** constants:

Constant	Meaning
B_THREAD_RUNNING	The thread is currently receiving attention from a CPU.
B_THREAD_READY	The thread is waiting for its turn to receive attention.
B_THREAD_SUSPENDED	The thread has been suspended or is freshly-spawned and is waiting to start.
B_THREAD_WAITING	The thread is waiting to acquire a semaphore. (Note that when a thread is sitting in a **wait_for_thread()** call, or is waiting to read from or write to a port, it's actually waiting to acquire a semaphore.) When in this state, the **sem** field of the **thread_info** structure is set to the **sem_id** number of the semaphore the thread is attempting to acquire.
B_THREAD_RECEIVING	The thread is sitting in a **receive_data()** function call.
B_THREAD_ASLEEP	The thread is sitting in a **snooze()** call.

The value of the priority field takes one of the following **long** constants (the difference between "time-sharing" priorities and "real-time" priorities is explained in "Thread Priority" on page 10):

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORITY	20

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	100
B_URGENT_PRIORITY	110
B_REAL_TIME_PRIORITY	120

Thread info is provided primarily as a debugging aid. None of the values that you find in a **thread_info** structure are guaranteed to be valid—the thread's state, for example, will almost certainly have changed by the time **get_thread_info()** returns.

Both functions return **B_NO_ERROR** upon success. If the designated thread isn't found—because *thread* in **get_thread_info()** isn't valid, or *n* in **get_nth_thread_info()** is out of range—the functions return **B_BAD_THREAD_ID**. If its *team* argument is invalid, **get_nth_thread_info()** return **B_BAD_TEAM_ID**.

See also: **get_team_info()**

has_data()

bool **has_data(**thread_id *thread***)**

Returns **TRUE** if the given thread has an unread message in its message cache, otherwise returns **FALSE**. Messages are sent to a thread's message cache through the **send_data()** call. To retrieve a message, you call **receive_data()**.

See also: **send_data()**, **receive_data()**

kill_team() *see* exit_thread()

kill_thread() *see* exit_thread()

receive_data()

```
long receive_data(thread_id *sender,
                  void *buffer,
                  long buffer_size)
```

Retrieves a message from the thread's message cache. The message will have been placed there through a previous **send_data()** function call. If the cache is empty, **receive_data()** blocks until one shows up—it never returns empty-handed.

The **thread_id** of the thread that called **send_data()** is returned by reference in the *sender* argument. Note that there's no guarantee that the sender will still be alive by the time you get its ID. Also, the value of *sender* going into the function is ignored—you can't ask for a message from a particular sender.

The **send_data()** function copies two pieces of data into a thread's message cache: A single four-byte code, and a arbitrarily long data buffer. The four-byte code is delivered, here, as **receive_data()**'s return value. The contents of the buffer part of the cache is copied into **receive_data()**'s *buffer* argument (you must allocate and free *buffer* yourself). The *buffer_size* argument tells the function how many bytes of data to copy. If you don't need the data buffer—if the code value returned directly by the function is sufficient—you set *buffer* to **NULL** and *buffer_size* to 0.

Unfortunately, there's no way to tell how much data is in the cache before you call **receive_data()**. If there's more data than *buffer* can accommodate, the unaccommodated portion is discarded—a second **receive_data()** call will not read the rest of the message. Conversely, if **receive_data()** asks for more data than was sent, the function returns with the excess portion of *buffer* unmodified—**receive_data()** doesn't wait for another **send_data()** call to provide more data with which to fill up the buffer.

Each **receive_data()** corresponds to exactly one **send_data()**. Lacking a previous invocation of its mate, **receive_data()** will block until **send_data()** is called. If you don't want to block, you should call **has_data()** before calling **receive_data()** (and proceed to **receive_data()** only if **has_data()** returns **TRUE**).

See also: **send_data()**, **has_data()**

rename_thread()

long **rename_thread(**thread_id *thread***,** const char **name***)**

Changes the name of the given thread to *name*. Keep in mind that the maximum length of a thread name is **B_OS_NAME_LENGTH** (32 characters).

If the *thread* argument isn't a valid **thread_id** number, **B_BAD_THREAD_ID** is returned. Otherwise, the function returns **B_NO_ERROR**.

resume_thread()

long **resume_thread(**thread_id *thread***)**

Tells a new or suspended thread to begin executing instructions. If the thread has just been spawned, its execution begins with the entry-point function (keep in mind that a freshly spawned thread doesn't run until told to do so through this function). If the thread was previously suspended (through **suspend_thread()**), it continues from where it was suspended.

This function only works on threads that have a status of **B_THREAD_SUSPENDED** (newly spawned threads are born with this state). You can't use this function to wake up a sleeping thread (**B_THREAD_ASLEEP**), or to unblock a thread that's waiting to acquire a semaphore (**B_THREAD_WAITING**) or waiting in a **receive_data()** call (**B_THREAD_RECEIVING**).

If the *thread* argument isn't a valid **thread_id** number, **B_BAD_THREAD_ID** is returned. If the thread exists but isn't suspended, **B_BAD_THREAD_STATE** is returned (the target thread is unaffected in this case). Otherwise, the function returns **B_NO_ERROR**.

See also: **wait_for_thread()**

send_data()

long **send_data(**thread_id *thread*,
long *code*,
void **buffer,*
long *buffer_size***)**

Copies data into *thread*'s message cache. The target thread can then retrieve the data from the cache by calling **receive_data()**. There are two parts to the data that you send:

- A single four-byte "code" given by the *code* argument.

- An arbitrarily long buffer of data that's pointed to by *buffer.* The length of the buffer, in bytes, is given by *buffer_size*.

If you only need to send the code, you should set *buffer* to **NULL** and *buffer_size* to 0. After **send_data()** returns you can free the *buffer* argument

Normally, **send_data()** returns immediately—it doesn't wait for the target to call **receive_data()**. However, **send_data()** will block if the target has an unread message from a previous **send_data()**—keep in mind that a thread's message cache is only one message deep. A thread that's blocked in **send_data()** assumes **B_THREAD_WAITING** status.

If the target thread couldn't allocate enough memory for its copy of *buffer*, this function fails and returns **B_NO_MEMORY**. If *thread* doesn't identify a valid thread, **BAD_THREAD_ID** is returned. Otherwise, the function succeeds and returns **B_NO_ERROR**.

See also: **receive_data()**, **has_data()**

set_thread_priority()

long **set_thread_priority(**thread_id *thread*, long *new_priority***)**

Resets the given thread's priority to *new_priority.* The priority level constants that are defined by the Kernel Kit are:

The value of the priority field takes one of the following **long** constants (the difference between "time-sharing" priorities and "real-time" priorities is explained in "Thread Priority" on page 10):

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORITY	20

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	100
B_URGENT_PRIORITY	110
B_REAL_TIME_PRIORITY	120

The difference between "time-sharing" priorities and "real-time" priorities is explained in "Thread Priority" on page 10.

If *thread* is invalid, **B_BAD_THREAD_ID** is returned. Otherwise, the priority to which the thread was set is returned.

snooze()

long **snooze(**double *microseconds***)**

Pauses the calling thread for the given number of microseconds. The thread's state is set to **B_THREAD_ASLEEP** while it's snoozing and restored to its previous state when it awakes.

The function returns **B_ERROR** if *microseconds* is less than 0.0, otherwise it returns **B_NO_ERROR**. Note that it isn't illegal to put a thread to sleep for 0.0 microseconds, but neither is it effectual; a call of **snooze(**0.0**)** is, essentially, ignored.

spawn_thread()

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

Creates a new thread and returns its **thread_id** identifier (a positive integer). The arguments are:

- *func* is a pointer to the thread's entry function. This is the function that the thread will execute when it's told to run.
- *name* is the name that you wish to give the thread. It can be, at most, **B_OS_NAME_LENGTH** (32) characters long.
- *priority* is the CPU priority level of the thread. It takes one of the following constant values (listed here from lowest to highest):

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORITY	20

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	100
B_URGENT_PRIORITY	110
B_REAL_TIME_PRIORITY	120

  For a complete explanation of these constants, see "Thread Priority" on page 10.

- *data* is forwarded as the argument to the thread's entry function.

A newly spawned thread is in a suspended state (**B_THREAD_SUSPENDED**). To tell the thread to run, you pass its **thread_id** to the **resume_thread()** function. The thread will continue to run until the entry-point function exits, or until the thread is explicitly killed (through a call to **exit_thread()**, **kill_thread()**, or **kill_team()**).

If all **thread_id** numbers are currently in use, **spawn_thread()** returns **B_NO_MORE_THREADS**; if the operating system lacks the memory needed to create the thread (which should be rare), **B_NO_MEMORY** is returned.

suspend_thread()

long **suspend_thread(**thread_id *thread***)**

Halts the execution of the given thread, but doesn't kill the thread entirely. The thread remains suspended until it is told to run through the **resume_thread()** function. Nothing prevents you from suspending your own thread, i.e.:

```
suspend_thread(find_thread(NULL));
```

Of course, this is only smart if you have some other thread that will resume you later.

This function only works on threads that have a status of **B_THREAD_RUNNING** or **B_THREAD_READY**. In other words, you can't suspend a thread that's sleeping, waiting to acquire a semaphore, waiting to receive data, or that's already suspended.

If the *thread* argument isn't a valid **thread_id** number, **B_BAD_THREAD_ID** is returned. If the thread exists, but is neither running nor ready to run, **B_BAD_THREAD_STATE** is returned. Otherwise, the function returns **B_NO_ERROR**.

wait_for_thread()

long **wait_for_thread(**thread_id *thread*, long **exit_value***)**

This function causes the calling thread to wait until *thread* (the "target thread") has died. If *thread* is suspended, the **wait_for_thread()** call will cause it to resume. Thus, you can use **wait_for_thread()** to tell a newly-spawned thread to start running.

When the target thread is dead, the value that was returned by its entry function (or that's imposed by **exit_thread()**, if such was called) is returned by reference in *exit_value*. If the target thread was killed (by **kill_thread()** or **kill_team()**), or if the entry function doesn't return a value, the value returned in *exit_value* will be unreliable.

If the target thread has already exited or is otherwise invalid, this function returns **B_BAD_THREAD_ID,** otherwise it returns **B_NO_ERROR.** Note that if the thread is killed while you're waiting for it, the function returns **B_NO_ERROR.**

See also: **resume_thread()**

Ports

Declared in: <kernel/OS.h>

Overview

A port is a system-wide message repository into which a thread can copy a buffer of data, and from which some other thread can then retrieve the buffer. This repository is implemented as a first-in/first-out message queue: A port stores its messages in the order in which they're received, and it relinquishes them in the order in which they're stored. Each port has its own message queue.

There are other ways to send data between threads. Most notably, the data-sending and -receiving mechanism provided by the **send_data()** and **receive_data()** functions can also transmit data between threads. But note these differences between using a port and using the **send_data()**/**receive_data()** functions:

- A port can hold more than one message at a time. A thread can only hold one at a time. Because of this, the function that writes data to a port (**write_port()**) rarely blocks. Sending data to a thread will block if the thread has a previous, unread message.
- The messages that are transmitted through a port aren't directed at a specific recipient—they're not addressed to a specific thread. A message that's been written to a port can be read by any thread. **send_data()**, by definition, has a specific thread as its target.

Creating a Port

A port is represented by a unique, system-wide **port_id** number (a positive integer). The **create_port()** function creates a new port and assigns it a **port_id** number. Although ports are accessible to all threads, the **port_id** numbers aren't disseminated by the operating system; if you create a port and want some other thread to be able to write to or read from it, you have to broadcast the **port_id** number to that thread. Typically, ports are used within a single team. The easiest way to broadcast a **port_id** number to the threads in a team is to declare it as a global variable.

A port is owned by the team in which it was created. When a team dies (when all its threads are killed, by whatever hand), the ports that belong to the team are deleted. A team can bestow ownership of its ports to some other team (through the **set_port_owner()** function).

If you want explicitly get rid of a port, you can call **delete_port()**. You can delete any port, not just those that are owned by the team of the calling thread.

The Message Queue: Reading and Writing Port Messages

The length of a port's message queue—the number of messages that it can hold at a time—is set when the port is created. The **B_MAX_PORT_COUNT** constant provides a reasonable queue length.

The functions **write_port()** and **read_port()** manipulate a port's message queue: **write_port()** places a message at the tail of the port's message queue; **read_port()** removes the message at the head of the queue and returns it the caller. **write_port()** blocks if the queue is full; it returns when room is made in the queue by an invocation of **read_port()**. Similarly, if the queue is empty, **read_port()** blocks until **write_port()** is called. When a thread is waiting in a **write_port()** or **read_port()** call, its state is **B_THREAD_SEM_WAIT** (it's waiting to acquire a system-defined, port-specific semaphore).

You can provide a timeout for your port-writing and port-reading operations by using the "full-blown" functions **write_port_etc()** and **read_port_etc()**. By supplying a timeout, you can ensure that your port operations won't block forever.

Although each port has its own message queue, all ports share a global "queue slot" pool—there are only so many message queue slots that can be used by all ports taken cumulatively. If too many port queues are allowed to fill up, the slot pool will drain, which will cause **write_port()** calls on less-than-full ports to block. To avoid this situation, you should make sure that your **write_port()** and **read_port()** calls are reasonably balanced.

The **write_port()** and **read_port()** functions are the only way to traverse a port's message queue. There's no notion of "peeking" at the queue's unread messages, or of erasing messages that are in the queue.

Port Messages

A port message—the data that's sent through a port—consists of a "message code" and a "message buffer." Either of these elements can be used however you like, but they're intended to fit these purposes:

- The message code (a single four-byte value) should be a mask, flag, or other predictable value that gives a general representation of the flavor or import of the message. For this to work, the sender and receiver of the message must agree on the meanings of the values that the code can take.

- The data in the message buffer can elaborate upon the code, identify the sender of the message, or otherwise supply additional information. The length of the buffer isn't restricted. To get the length of the message buffer that's at the head of a port's queue, you call the **port_buffer_size()** function.

The message that you pass to **write_port()** is copied into the port. After **write_port()** returns, you may free the message data without affecting the copy that the port holds.

When you read a port, you have to supply a buffer into which the port mechanism can copy the message. If the buffer that you supply isn't large enough to accommodate the message, the unread portion will be lost—the next call to **read_port()** won't finish reading the message.

You typically allocate the buffer that you pass to **read_port()** by first calling **port_buffer_size()**, as shown below:

```
char *buf;
long size;
long code;

/* We'll assume that my_port is valid.
 * port_buffer_size() will block until a message shows up.
 */
if ((size = port_buffer_size(my_port) < B_NO_ERROR)
    /* Handle the error */

if (size > 0)
    buf = (char *)malloc(size * sizeof(char));
else
    buf = 0;

/* Now we can read the buffer. */
if (read_port(my_port, &code, (void *)buf, size) < B_NO_ERROR)
    /* Handle the error */
```

Obviously, there's a race condition (in the example) between **port_buffer_size()** and the subsequent **read_port()** call—some other thread could read the port in the interim. If you're going to use **port_buffer_size()** as shown in the example, you shouldn't have more than one thread reading the port.

As stated in the example, **port_buffer_size()** blocks until a message shows up. If you don't want to (potentially) block forever, you should use the **port_buffer_size_etc()** version of the function. As with the other **...etc()** functions, **port_buffer_size_etc()** provides a timeout option.

Function Descriptions

create_port()

port_id **create_port(**long *queue_length*, const char **name***)**

Creates a new port and returns its **port_id** number. The port's name is set to *name* and the length of its message queue is set to *queue_length*. Neither the name nor the queue length can be changed once they're set. The name shouldn't exceed **B_OS_NAME_LENGTH** (32) characters.

In setting the length of a port's message queue, you're telling it how many messages it can hold at a time. When the queue is filled—when it's holding *queue_length* messages—subsequent invocations of **write_port()** (on that port) block until room is made in the queue (through calls to **read_port()**) for the additional messages. As a convenience, you can use the **B_MAX_PORT_COUNT** constant as the *queue_length* value; this constant represents the (ostensible) maximum port queue length. Once the queue length is set (here), it can't be changed.

This function also sets the owner of the port to be the team of the calling thread. Ownership can subsequently be transferred through the **set_port_owner()** function. When a port's owner dies (when all the threads in the team are dead), the port is automatically deleted. If you want to delete a port prior to its owner's death, use the **delete_port()** function.

The function returns **B_BAD_VALUE** if *queue_length* is out of bounds (less than one or greater than the maximum capacity). It returns **B_NO_MORE_PORTS** if all **port_id** numbers are currently being used.

See also: **delete_port()**, **set_port_owner()**

delete_port()

long **delete_port(**port_id *port***)**

Deletes the given port. The port's message queue doesn't have to be empty—you can delete a port that's holding unread messages. Threads that are blocked in **read_port()** or **write_port()** calls on the port are automatically unblocked (and return **B_BAD_SEM_ID**).

The thread that calls **delete_port()** doesn't have to be a member of the team that owns the port; any thread can delete any port.

The function returns **B_BAD_PORT_ID** if *port* isn't a valid port; otherwise it returns **B_NO_ERROR**.

See also: **create_port()**

find_port()

port_id **find_port(**const char **port_name***)**

Returns the **port_id** of the named port. If the argument doesn't name an existing port, **B_NAME_NOT_FOUND** is returned.

See also: **create_port()**

get_port_info(), get_nth_port_info()

long **get_port_info(**port_id *port*, port_info **info***)**
long **get_nth_port_info(**team_id *team*, long *n*, port_info **info***)**

These functions copy, into the *info* argument, the **port_info** structure for a particular port:

- The **get_port_info()** function gets this information for the port identified by *port*.
- The **get_nth_port_info()** function retrieves port information for the *n*'th port (zero-based) that's owned by the team identified by *team*. If *team* is 0 (zero), all teams are considered. You use this function to retrieve the info structures of all the ports in a team (or in all teams) by repeatedly calling the function with a monotonically increasing value of *n*—the actual value of *n* has no other significance. When, in this scheme, the function no longer returns **B_NO_ERROR**, all candidate ports will have been visited.

The **port_info** structure is defined as:

```
typedef struct port_info {
      port_id port;
      team_id team;
      char name[B_OS_NAME_LENGTH];
      long capacity;
      long queue_count;
      long total_count;
} port_info
```

The structure's fields are:

- **port**. The **port_id** number of the port.
- **team**. The **team_id** of the port's team.
- **name**. The name assigned to the port.
- **capacity**. The length of the port's message queue.
- **queue_count**. The number of messages currently in the queue.
- **total_count**. The total number of message that have been read from the port.

Note that the **total_count** number doesn't include the messages that are currently in the queue.

The information in the **port_info** structure is guaranteed to be internally consistent, but the structure as a whole should be consider to be out-of-date as soon as you receive it. It provides a picture of a port as it exists just before the info-retrieving function returns.

The functions return **B_NO_ERROR** if the designated port is successfully found. Otherwise, they return **B_BAD_PORT_ID**, **B_BAD_TEAM_ID**, or **B_BAD_INDEX**.

port_buffer_size(), port_buffer_size_etc()

long **port_buffer_size(**port_id *port***)**

long **port_buffer_size_etc(**port_id *port*, long *flags*, double *timeout***)**

These functions return the length (in bytes) of the message buffer that's at the head of *port*'s message queue. You call this function in order to allocate a sufficiently large buffer in which to retrieve the message data.

The **port_buffer_size()** function blocks if the port is currently empty. It unblocks when a **write_port()** call gives this function a buffer to measure (even if the buffer is 0 bytes long), or when the port is deleted.

The **port_buffer_size_etc()** function lets you set a limit on the amount of time the function will wait for a message to show up. To set the limit, you pass **B_TIMEOUT** as the flags argument, and set *timeout* to the amount of time, in microseconds, that you're willing to wait.

If *port* doesn't identify an existing port (or if the port is deleted while the function is blocked), **B_BAD_PORT_ID** is returned. If the *timeout* limit is exceeded, **B_TIMED_OUT** is returned. If the *timeout* limit is 0.0 (and **B_TIMEOUT** is set), and there are no messages in the queue, the function immediately returns **B_WOULD_BLOCK**.

See also: **read_port()**

port_count()

long **port_count(**port_id *port***)**

Returns the number of messages that are currently in *port*'s message queue. This is the number of messages that have been written to the port through calls to **write_port()** but that haven't yet been picked up through corresponding **read_port()** calls. This function is provided mostly as a convenience and a semi-accurate debugging tool. The value that it returns is inherently undependable (there's no guarantee that additional **read_port()** or **write_port()** calls won't change the count as this function is returning).

If *port* isn't a valid port identifier, **B_BAD_PORT_ID** is returned.

See also: **get_port_info()**

read_port(), read_port_etc()

```
long read_port(port_id port,
               long *msg_code,
               void *msg_buffer,
               long buffer_size)

long read_port_etc(port_id port,
                   long *msg_code,
                   void *msg_buffer,
                   long buffer_size,
                   long flags,
                   double timeout)
```

These functions remove the message at the head of *port*'s message queue and copy the messages's contents into the *msg_code* and *msg_buffer* arguments. The size of the *msg_buffer* buffer, in bytes, is given by *buffer_size*. It's up to the caller to ensure that the message buffer is large enough to accommodate the message that's being read. If you want a hint about the message's size, you should call **port_buffer_size()** before calling this function.

If *port*'s message queue is empty when you call **read_port()**, the function will block. It returns when some other thread writes a message to the port through **write_port()**. A blocked read is also unblocked if the port is deleted.

The **read_port_etc()** function lets you set a limit on the amount of time the function will wait for a message to show up. To set the limit, you pass **B_TIMEOUT** as the flags argument, and set *timeout* to the amount of time, in microseconds, that you're willing to wait.

The functions returns **B_BAD_PORT_ID** if *port* isn't valid (this includes the case where the port is deleted during a blocked **read_port()** call). If the *timeout* value is exceeded, **B_TIMED_OUT** is returned. If the *timeout* limit is 0.0 (with **B_TIMEOUT** set), and there are no messages in the queue, the function immediately returns **B_WOULD_BLOCK**.

A successful call returns the number of bytes that were written into the *msg_buffer* argument.

See also: **write_port()**, **port_buffer_size()**

set_port_owner()

```
long set_port_owner(port_id port, team_id team)
```

Transfers ownership of the designated port to *team*. A port can only be owned by one team at a time; by setting a port's owner, you remove it from its current owner.

There are no restrictions on who can own a port, or on who can transfer ownership. In other words, the thread that calls **set_port_owner()** needn't be part of the team that currently owns the port, nor must you only assign ports to the team that owns the calling thread (although these two are the most likely scenarios).

Port ownership is meaningful for one reason: When a team dies (when all its threads are dead), the ports that are owned by that team are deleted. Ownership, otherwise, has no significance—it carries no special privileges or obligations.

To discover a port's owner, use the **get_port_info()** function.

set_port_owner() fails and returns **B_BAD_PORT_ID** or **B_BAD_TEAM_ID** if one or the other argument is invalid. Otherwise it returns **B_NO_ERROR**.

See also: **get_port_info()**

write_port(), write_port_etc()

```
long write_port(port_id port,
                long msg_code,
                void *msg_buffer,
                long buffer_size)

long write_port_etc(port_id port,
                long msg_code,
                void *msg_buffer,
                long buffer_size,
                long flags,
                double timeout)
```

These functions place a message at the tail of *port*'s message queue. The message consists of *msg_code* and *msg_buffer*:

- *msg_code* holds the message code. This is a mask, flag, or other predictable value that gives a general representation of the message.
- *msg_buffer* is a pointer to a buffer that can be used to supply additional information. You pass the length of the buffer, in bytes, as the value of the *buffer_size* argument. The buffer can be arbitrarily long.

If the port's queue is full when you call **write_port()**, the function will block. It returns when a **read_port()** call frees a slot in the queue for the new message. A blocked **write_port()** will also return if the target port is deleted.

The **write_port_etc()** function lets you set a limit on the amount of time the function will wait for a free queue slot. To set the limit, you pass **B_TIMEOUT** as the flags argument, and set *timeout* to the amount of time, in microseconds, that you're willing to wait.

If *port* isn't valid **B_BAD_PORT_ID** is returned (this includes the case where the port is deleted during a blocked **read_port()** call). If the *timeout* value is exceeded, **B_TIMED_OUT** is returned. If the *timeout* limit is 0.0 (with **B_TIMEOUT** set), and the target port's queue is full, the function immediately returns **B_WOULD_BLOCK**. A successful call returns **B_NO_ERROR**.

See also: **read_port()**

Semaphores

Declared in: <kernel/OS.h>

Overview

A semaphore is a token that's used in a multi-threaded operating system to coordinate access, by competing threads, to "protected" resources or operations. This coordination usually takes one of these tacks:

- The most common use of semaphores is to limit the number of threads that can execute a piece of code at the same time. The typical limit is one—in other words, semaphores are most often used to create mutually exclusive locks.
- Semaphores can also be used to impose the order in which a series of interdependent operations are performed.

Examples of these uses are given in sections below.

How Semaphores Work

A semaphore acts as a key that a thread must acquire in order to continue execution. Any thread that can identify a particular semaphore can attempt to acquire it by passing its **sem_id** identifier—a system-wide number that's assigned when the semaphore is created—to the **acquire_sem()** function. The function doesn't return until the semaphore is actually acquired. (An alternate function, **acquire_sem_etc()** lets you specify a limit, in microseconds, on the amount of time you're willing to wait for the semaphore to be acquired. Unless otherwise noted, characteristics ascribed to **acquire_sem()** apply to **acquire_sem_etc()** as well.)

When a thread acquires a semaphore, that semaphore (typically) becomes unavailable for acquisition by other threads (in the rarer case, more than one thread is allowed to acquire the semaphore at a time; the precise determination of availability is explained in "The Thread Count" on page 32). The semaphore remains unavailable until it's passed in a call to the **release_sem()** function.

The code that a semaphore "protects" lies between the calls to **acquire_sem()** and **release_sem()**. The disposition of these functions in your code usually follows this pattern:

```
acquire_sem(my_semaphore);
/* Protected code goes here. */
release_sem(my_semaphore);
```

Keep in mind that these function calls needn't be so explicitly balanced. A semaphore can be acquired within one function and released in another. Acquisition and release of the same semaphore can even be performed by two different threads; an example of this is given in "Using Semaphores to Impose an Execution Order" on page 35.

The Thread Queue

Every semaphore has its own *thread queue*: This is a list that identifies the threads that are waiting to acquire the semaphore. A thread that attempts to acquire an unavailable semaphore is placed at the tail of the semaphore's thread queue; from the programmer's point of view, a thread that's been placed in the queue will be blocked in the **acquire_sem()** call. Each call to **release_sem()** "releases" the thread at the head of that semaphore's queue (if there are any waiting threads), thus allowing the thread to return from its call to **acquire_sem()**.

Semaphores don't discriminate between acquisitive threads—they don't prioritize or otherwise reorder the threads in their queues—the oldest waiting thread is always the next to acquire the semaphore.

The Thread Count

To assess availability, a semaphore looks at its *thread count*. This is a counting variable that's initialized when the semaphore is created. The ostensible (although, as we shall see, not entirely accurate) meaning of a thread count's initial value, which is passed as the first argument to **create_sem()**, is the number of threads that can acquire the semaphore at a time. For example, a semaphore that's used as a mutually exclusive lock takes an initial thread count of 1—in other words, only one thread can acquire the semaphore at a time.

Calls to **acquire_sem()** and **release_sem()** alter the semaphore's thread count: **acquire_sem()** decrements the count, and **release_sem()** increments it. When you call **acquire_sem()**, the function looks at the thread count (before decrementing it) to determine if the semaphore is available:

- If the count is greater than zero, the semaphore is available for acquisition, so the function returns immediately.
- If the count is zero or less, the semaphore is unavailable, and so the thread is placed in the semaphore's thread queue.

The initial thread count isn't an inviolable limit on the number of threads that can acquire a given semaphore—it's simply the initial value for the sempahore's thread count variable. For example, if you create a semaphore with an initial thread count of 1 and then immediately call **release_sem()** five times, the semaphore's thread count will increase to 6. Furthermore, although you can't initialize the thread count to less-than-zero, an initial value of zero itself is common—it's an integral part of using semaphores to impose an execution order (as demonstrated later).

Summarizing the description above, there are three significant thread count value ranges:

- A positive thread count (*n*) means that there are no threads in the semaphore's queue, and the next *n* **acquire_sem()** calls will return without blocking.
- If the count is 0, there are no queued threads, but the next **acquire_sem()** call will block.
- A negative count (-*n*) means there are *n* threads in the semaphore's thread queue, and the next call to **acquire_sem()** will block.

Although it's possible to retrieve the value of a semaphore's thread count (by looking at a field in the semaphore's **sem_info** structure, as described later), you should only do so for amusement—while you're debugging, for example. You should never predicate your code on the basis of a semaphore's thread count.

Using a Semaphore as a Lock

As mentioned above, the most common use of semaphores is to ensure that only one thread is executing a certain piece of code at a time. The following example demonstrates this use.

Consider an application that manages a one-job-at-a-time device such as a printer. When the application wants to start a new print job (upon a request from some other application, no doubt) it spawns and runs a thread to perform the actual data transmission. Given the nature of the device, each spawned thread must be allowed to complete its transmission before the next thread takes over. However, your application wants to accept print requests (and so spawn threads) as they arrive.

To ensure that the spawned threads don't interrupt each other, you can define a semaphore that's acquired and released—that, in essence, is "locked" and "unlocked"—as a thread begins and ends its transmission, as shown below. The thread functions that are used in the example are described in "Threads and Teams" on page 5.

```
/* Include the semaphore API declarations. */
#include <OS.h>

/* The semaphore is declared globally so the spawned threads
 * will be able to get to it (there are other ways of
 * broadcasting the sem_id, but this is the easiest).
 */
sem_id print_sem;

/* print_something() is the data-transmission function.
 * The data itself would probably be passed as an argument
 * (which isn't shown in this example).
 */
long print_something(void *data);
```

```
main()
{
    /* Create the semaphore with an initial thread count of 1.
     * If the semaphore can't be created (error conditions
     * are listed later), we exit.  The second argument to
     * create_sem(), as explained in the function
     * descriptions is a handy string name for the semaphore.
     */
    if ((print_sem = create_sem(1, "print sem")) < B_NO_ERROR)
        exit -1;

    while (1)
    {
        /* Wait-for-a-request code and break conditions
         * go here.
         */
        ...

        /* Spawn a thread that calls print_something(). */
        if (resume_thread(spawn_thread(print_something ...))
            < B_NO_ERROR)
            break;
    }

    /* Acquire the semaphore and delete it (as explained
     * later)
     */
    acquire_sem(print_sem);
    delete_sem(print_sem);
    exit 0;
}

long print_something(void *data)
{
    /* Acquire the semaphore; an error means the semaphore
     * is no longer valid.  And we'll just die if it's no good.
     */
    if (acquire_sem(print_sem) < B_NO_ERROR)
        return 0;

    /* The code that sends data to the printer goes here. */

    /* Release the semaphore. */
    release_sem(print_sem);

    return 0;
}
```

The **acquire_sem()** and **release_sem()** calls embedded in the **print_something()** function "protect" the data-transmission code. Although any number of threads may concurrently execute **print_something()**, only one at a time is allowed to proceed past the **acquire_sem()** call.

Deleting a Semaphore

Notice that the example explicitly deletes the **print_sem** semaphore before it exits. This isn't wholly necessary: Every semaphore is owned by a team (the team of the thread that called **create_sem()**). When the last thread in a team dies, it takes the team's semaphores with it.

Prior to the death of a team, you can explicitly delete a semaphore through the **delete_sem()** call. Note, however, that **delete_sem()** must be called from a thread that's a member of the team that owns the semaphore—you can't delete another team's semaphores.

You're allowed to delete a semaphore even if it still has threads in its queue. However, you usually want to avoid this, so deleting a semaphore may require some thought. In the example, the main thread (the thread that executes the **main()** function) makes sure all print threads have finished by acquiring the semaphore before deleting it. When the main thread is allowed to continue (when the **acquire_sem()** call returns) the queue is sure to be empty and all print jobs will have completed.

When you delete a semaphore (or when it dies naturally), all its queued threads are immediately allowed to continue—they all return from **acquire_sem()** at once. You can distinguish between a "normal" acquisition and a "semaphore deleted" acquisition by the value that's returned by **acquire_sem()** (the specific return values are listed in the function descriptions, below).

Using Semaphores to Impose an Execution Order

Semaphores can also be used to coordinate threads that are performing separate operations, but that need to perform these operations in a particular order. In the following example, an application repeatedly spawns, in no particular order, threads that either write to or read from a global buffer. Each writing thread must complete before the next reading thread starts, and each written message must be fully read exactly once. Thus, the two operations must alternate (with a writing thread going first). Two semaphores are used to coordinate the threads that perform these operations:

```
/* Here's the global buffer. */
char buf[1024];

/* The ok_to_read and ok_to_write semaphores inform the
 * appropriate threads that they can proceed.
 */
sem_id ok_to_write, ok_to_read;

/* These are the writing and reading functions. */
long write_it(void *data);
long read_it(void *data);
```

```
main()
{
    /* These will be used when we delete the semaphores. */
    long write_count, read_count;

    /* Create the semaphores.  ok_to_write is created with a
     * thread count of 1; ok_to_read's count is set to 0.
     * This is explained below.
     */
    if ((ok_to_write = create_sem(1, "write sem"))<B_NO_ERROR)
        return (B_ERROR);

    if ((ok_to_read = create_sem(0, "read sem")) < B_NO_ERROR)
    {
        delete_sem(ok_to_write);
        return (B_ERROR);
    }

    bzero(buf,1024);

    /* Spawn some reading and writing threads. */
    while(1)
    {
        if ( ... ) /* spawn-a-writer condition */
            resume_thread(spawn_thread(write_it, ...));
        if ( ... ) /* spawn-a-reader condition */
            resume_thread(spawn_thread(read_it, ...);
        if ( ... ) /* break condition */
            break;
    }

    /* It's time to delete the semaphores.  First, get the
     * semaphores' thread counts.
     */
    if (get_sem_count(ok_to_write, &write_count) < B_NO_ERROR)
    {
        delete_sem(ok_to_read);
        return (B_ERROR);
    }

    if (get_sem_count(ok_to_read, &read_count) < B_NO_ERROR)
    {
        delete_sem(ok_to_write);
        return (B_ERROR);
    }

    /* Place this thread at the end of whichever queue is
     * shortest (or the writing queue if they're equal).
     * Remember: thread count is decremented as threads
     * are placed in the queue, so the shorter queue is
     * the one with the greater thread count.
     */
    if (write_count >= read_count)
        acquire_sem(ok_to_write);
        else
            acquire_sem(ok_to_read);

        /* Delete the semaphores and exit. */
        delete_sem(ok_to_write);
        delete_sem(ok_to_read);
        return (B_NO_ERROR);
    }

    long write_it(void *data)
    {
        /* Acquire the writing semaphore. */
        if (acquire_sem(ok_to_write) < B_NO_ERROR)
            return (B_ERROR);

        /* Write to the buffer. */
        strncpy(buf, (char *)data, 1023);

        /* Release the reading semaphore. */
        return (release_sem(ok_to_read));
    }

    long read_it(void *data)
    {
        /* Acquire the reading semaphore. */
        if (acquire_sem(ok_to_read) < B_NO_ERROR)
            return (B_ERROR);

        /* Read the message and do something with it. */
        ...

        /* Release the writing semaphore. */
        return (release_sem(ok_to_write));
    }
```

Notice the distribution of the **acquire_sem()** and **release_sem()** calls for the respective semaphores: The writing function acquires the writing semaphore (*ok_to_write*) and then releases the reading semaphore (*ok_to_read*). The reading function does the opposite. Thus, after the buffer has been written to, no other thread can write to it until it has been read (and vice versa).

By setting *ok_to_write*'s initial thread count to 1 and *ok_to_read*'s initial thread count to 0, you ensure that a writing operation will be performed first. If a reading thread is spawned first, it will block until a writing thread releases the *ok_to_read* semaphore.

When it's semaphore-deletion time in the example, the main thread acquires one of the semaphores. Specifically, it acquires the semaphore that has the fewer threads in its queue. This allows the remaining (balanced) pairs of reading and writing threads to complete before the semaphores are deleted, and throws away any unpaired reading or writing threads. (Actually, the unpaired threads aren't "thrown away" as the semaphore upon which they're waiting is deleted, but by the error check in the first line of the reading or writing function. As mentioned earlier, deleting the semaphore releases its queued threads, allowing them, in this instance, to rush to their deaths.)

Broadcasting Semaphores

The **sem_id** number that identifies a semaphore is a system-wide token—the **sem_id** values that you create in your application will identify your semaphores in all other applications as well. It's possible, therefore, to broadcast the **sem_id** numbers of the semaphores that you create and so allow other applications to acquire and release them—but it's not a very good idea. A semaphore is best controlled if it's created, acquired, released, and deleted within the same team. If you want to provide a protected service or resource to other applications, you should follow the model used by the examples: Your application should accept messages from other applications and then spawn threads that acquire and release the appropriate semaphores.

Functions

acquire_sem(), acquire_sem_etc()

long **acquire_sem(**sem_id *sem***)**
long **acquire_sem_etc(**sem_id *sem***,** long *count*, long *flags*, double *timeout***)**

These functions attempt to acquire the semaphore identified by the *sem* argument. Except in the case of an error, **acquire_sem()** doesn't return until the semaphore has actually been acquired.

acquire_sem_etc() is the full-blown acquisition version: It's essentially the same as **acquire_sem()**, but, in addition, it lets you acquire a semaphore more than once, and also provides a timeout facility:

- The *count* argument lets you specify that you want the semaphore to be acquired *count* times. This means that the semaphore's thread count is decremented by the specified amount. It's illegal to specify a count that's less than 1.
- To enable the timeout, you pass **B_TIMEOUT** as the *flags* argument, and set *timeout* to the amount of time, in microseconds, that you're willing to wait for the semaphore to be acquired. If the semaphore hasn't been acquired within *timeout* microseconds, the function gives up and returns the value **B_TIMED_OUT**. If you specify a *timeout* of 0.0 and the semaphore isn't immediately available, the function returns **B_WOULD_BLOCK**.

In addition to **B_TIMEOUT**, the Kernel Kit defines two other semaphore-acquisition flag constants (**B_CAN_INTERRUPT** and **B_CHECK_PERMISSION**). These additional flags are used by device drivers—adding these flags into a "normal" (or "user-level") acquisition has no effect. However, you should be aware that the **B_CHECK_PERMISSION** flag is always added in to user-level semaphore acquisition in order to protect system-defined semaphores.

Other than the timeout and the acquisition count, there's no difference between the two acquisition functions. Specifically, any semaphore can be acquired through either of these functions; you always release a semaphore through **release_sem()** (or **release_sem_etc()**) regardless of which function you used to acquire it.

To determine if the semaphore is available, the function looks at the semaphore's thread count (before decrementing it):

- If the thread count is positive, the semaphore is available and the current acquisition succeeds. The **acquire_sem()** or **acquire_sem_timeout()** function returns immediately upon acquisition.

- If the thread count is zero or less, the calling thread is placed in the semaphore's thread queue where it waits for a corresponding **release_sem()** call to de-queue it (or for the timeout to expire).

If the *sem* argument doesn't identify a valid semaphore, **B_BAD_SEM_ID** is returned. It's possible for a semaphore to become invalid while an acquisitive thread is waiting in the semaphore's queue. For example, if your thread calls **acquire_sem()** on a valid (but unavailable) semaphore, and then some other thread deletes the semaphore, your thread will return **B_BAD_SEM_ID** from its call to **acquire_sem()**.

If you pass an illegal *count* value (less than 1) to **acquire_sem_etc()**, the function returns **B_BAD_VALUE**. If the acquisition time surpasses the designated timeout limit (with **B_TIMEOUT** set), the **acquire_sem_etc()** function returns **B_TIMED_OUT**; if the timeout value is 0.0 and the semaphore isn't immediately available, the function returns **B_WOULD_BLOCK**.

If the semaphore is successfully acquired, the functions return **B_NO_ERROR**.

See also: **release_sem()**

create_sem()

sem_id **create_sem(**long *thread_count*, const char **name***)**

Creates a new semaphore and returns a system-wide **sem_id** number that identifies it. The arguments are:

- *thread_count* initializes the semaphore's *thread count*, the counting variable that's decremented and incremented as the semaphore is acquired and released (respectively). You can pass any non-negative number as the count, but you typically pass either 1 or 0, as demonstrated in the examples above.

- *name* is an optional string name that you can assign to the semaphore. The name is meant to be used only for debugging. A semaphore's name needn't be unique—any number of semaphores can have the same name.

Valid **sem_id** numbers are positive integers. You should always check the validity of a new semaphore through a construction such as

```
if ((my_sem = create_sem(1,"My Semaphore")) < B_NO_ERROR)
```

```
        /* If it's less than B_NO_ERROR, my_sem is invalid. */
```

create_sem() sets the new semaphore's owner to the team of the calling thread. Ownership may be re-assigned through the **set_sem_owner()** function. When the owner dies (when all the threads in the team are dead), the semaphore is automatically deleted. The owner is also signficant in a **delete_sem()** call: Only those threads that belong to a semaphore's owner are allowed to delete that semaphore.

The function returns one of the following codes if the semaphore couldn't be created:

Return Code	Meaning
B_BAD_ARG_VALUE	Invalid *thread_count* value (less than zero).
B_NO_MEMORY	Not enough memory to allocate the semaphore's name.
B_NO_MORE_SEMS	All valid **sem_id** numbers are being used.

See also: **delete_sem()**

delete_sem()

long **delete_sem(**sem_id *sem***)**

Deletes the semaphore identified by the argument. If there are any threads waiting in the semaphore's thread queue, they're immediately de-queued and allowed to continue execution.

This function may only be called from a thread that belongs to the target semaphore's owner; if the calling thread belongs to a different team, or if *sem* is invalid, the function returns **B_BAD_SEM_ID**. Otherwise, it returns**B_NO_ERROR**.

See also: **acquire_sem(**)

get_sem_count()

long **get_sem_count(**sem_id *sem,* long **thread_count***)**

Returns, by reference in *thread_count*, the value of the semaphore's thread count variable:

- A positive thread count (*n*) means that there are no threads in the semaphore's queue, and the next *n* **acquire_sem()** calls will return without blocking.
- If the count is zero, there are no queued threads, but the next **acquire_sem()** call will block.
- A negative count (-*n*) means there are *n* threads in the semaphore's thread queue and the next call to **acquire_sem()** will block.

By the time this function returns and you get a chance to look at the *thread_count* value, the semaphore's thread count may have changed. Although watching the thread count might help you while you're debugging your program, this function shouldn't be an integral part of the design of your application.

If *sem* is a valid semaphore identifier, the function returns **B_NO_ERROR**; otherwise, **B_BAD_SEM_ID** is returned (and the value of the *thread_count* argument that you pass in isn't changed).

See also: **get_sem_info()**

get_sem_info(), get_nth_sem_info()

long **get_sem_info(**sem_id *sem*, sem_info **info***)**
long **get_nth_sem_info(**team_id *team*, long *n*, sem_info **info***)**

These functions copy, into the *info* argument, the **sem_info** structure for a particular semaphore:

- The **get_sem_info()** function gets this information for the semaphore identified by *sem*.
- The **get_nth_sem_info()** function retrieves semaphore information for the *n*'th semaphore (zero-based) that's owned by the team identified by *team*. If *team* is 0 (zero), all teams are considered. You use this function to retrieve the info structures of all the semaphores in a team (or in all teams) by repeatedly calling the function with a monotonically increasing value of *n*—the actual value of *n* has no other significance. When, in this scheme, the function no longer returns **B_NO_ERROR**, all candidate semaphores will have been visited.

The **sem_info** structure is defined as:

```
typedef struct sem_info {
        sem_id sem;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        long count;
        thread_id latest_holder;
} sem_info
```

The structure's fields are:

- **sem**. The **sem_id** number of the semaphore.
- **team**. The **team_id** of the semaphore's owner.
- **name**. The name assigned to the semaphore.
- **count**. The semaphore's thread count.
- **latest_holder**. The thread that most recently acquired the semaphore.

Note that the thread that's identified in the **lastest_holder** field may no longer be holding the semaphore—it may have since released the semaphore. The latest holder is simply the last thread to have called **acquire_sem()** (of whatever flavor) on this semaphore.

The information in the **sem_info** structure is guaranteed to be internally consistent, but the structure as a whole should be consider to be out-of-date as soon as you receive it. It provides a picture of a semaphore as it exists just before the info-retrieving function returns.

The functions return **B_NO_ERROR** if the designated semaphore is successfully found. Otherwise, they return **B_BAD_SEM_ID**, **B_BAD_TEAM_ID**, or **B_BAD_INDEX**.

release_sem(), release_sem_etc()

long **release_sem(**sem_id *sem***)**
long **release_sem_etc(**sem_id *sem*, long *count*, long *flags***)**

The **release_sem()** function de-queues the thread that's waiting at the head of the semaphore's thread queue (if any), and increments the semaphore's thread count. **release_sem_etc()** does the same, but for *count* threads.

Normally, releasing a semaphore automatically invokes the kernel's scheduler. In other words, when your thread calls **release_sem()** (or the sequel), you're pretty much guaranteed that some other thread will be switched in immediately afterwards, even if your thread hasn't gotten its fair share of CPU time. If you want to subvert this automatism, call **release_sem_etc()** with a *flags* value of **B_DO_NOT_RESCHEDULE**. Preventing the automatic rescheduling is particularly useful if you're releasing a number of different semaphores all in a row: By avoiding the rescheduling you can prevent some unnecessary context switching.

If *sem* is a valid semaphore identifier, these functions return **B_NO_ERROR**; if it's invalid, they return **B_BAD_SEM_ID**. Note that if a released thread deletes the semaphore (before the releasing function returns), these functions will still return **B_NO_ERROR**.

The *count* argument to **release_sem_count()** must be greater than zero; the function returns **B_BAD_VALUE** otherwise.

See also: **acquire_sem()**

set_sem_owner()

long **set_sem_owner(**sem_id *sem*, team_id *team***)**

Transfers ownership of the designated semaphore to *team*. A semaphore can only be owned by one team at a time; by setting a semaphore's owner, you remove it from its current owner.

There are no restrictions on who can own a semaphore, or on who can transfer ownership. In practice, however, the only reason you should ever transfer ownership is if you're writing a device driver and you need to bequeath a semaphore to the kernel (the team of which is known, for this purpose, as **B_SYSTEM_TEAM**).

Semaphore ownership is meaningful for two reason: When a team dies (when all its threads are dead), the semaphores that are owned by that team are deleted. Also, only a thread that belongs to a semaphore's owner is allowed to delete that semaphore.

To discover a semaphore's owner, use the **get_sem_info()** function.

set_sem_owner() fails and returns **B_BAD_SEM_ID** or **B_BAD_TEAM_ID** if one or the other argument is invalid. Otherwise it returns **B_HOKEY_POKEY**.

See also: **get_sem_info()**

Areas

Declared in: <kernel/OS.h>

Overview

An area is a chunk of virtual memory. As such, it has all the expected properties of virtual memory: It has a starting address, a size, the addresses it comprises are contiguous, and it maps to (possibly non-contiguous) physical memory. The primary differences between an area and “standard” virtual memory (memory that you allocate through **malloc()**, for example) are these:

- Different areas can refer to the same physical memory. Put another way, different virtual memory addresses can map to the same physical locations. Furthermore, the different areas needn’t belong to the same application. By creating and “cloning” areas, applications can easily share the same data.
- You can specify that the area’s physical memory be locked into RAM when it’s created, locked on a page-by-page basis as pages are swapped in, or that it be swapped in and out as needed.
- Areas always start on a page boundary, and are allocated in integer multiples of the size of a page. (A page is 4096 bytes, as represented by the **B_PAGE_SIZE** constant.)
- You can specify the starting address of the area’s virtual memory. The specification can require that the area start precisely at a certain address, anywhere above a certain address, or anywhere at all.
- An area can be read- and write-protected.

Because areas are large—4096 bytes minimum—you don’t create them arbitrarily. The two most compelling reasons to create an area are the two first points listed above: To share data among different applications, and to lock memory into RAM.

Identifying an Area

An area is uniquely identified (system-wide) by its **area_id** number. The **area_id** is assigned automatically by **create_area()**, a function that does what it says. Most of the other area functions require an **area_id** argument.

When you create an area, you get to name it. Area names are not unique—any number of areas can be assigned the same name.

Sharing Areas

If you want to share an area with another application, you can broadcast the area's **area_id** number, but it's recommended that, instead, you publish the area's name. Given an area name, a "remote" application can retrieve the area's ID number by calling **find_area()**.

To use an area that was created by another application, the first thing you do, having acquired the area's **area_id** through **find_area()**, is "clone" the area. You do this by calling the **clone_area()** function. The function returns a new **area_id** number that identifies your clone of the original area. All further references to the area (in the cloning application) must be based on the ID of the clone.

The physical memory that lies beneath a cloned area is never implicitly copied—for example, the area mechanism doesn't perform a "copy-on-write." If two areas (more specifically, two **area_id** numbers) refer to the same memory because of cloning, a data modification that's affected through one area will be seen by the other area.

Note: Because names aren't unique, multiple calls to **find_area()** with the same name won't all necessarily return the same **area_id**—consider the case where more than one instantiation of the same area-creating application is running on your computer.

Locking an Area

When you're working with moderately large amounts of data, it's often the case that you would prefer that the data remain in RAM, even if the rest of your application needs to be swapped out. An argument to **create_area()** lets you declare, through the use of one of the following constants, the locking scheme that you wish to apply to your area:

- **B_FULL_LOCK** means the area's memory is locked into RAM when the area is created, and won't be swapped out.
- **B_LAZY_LOCK** allows individual pages of memory to be brought into RAM through the natural order of things and *then* locks them.
- **B_NO_LOCK** means pages are never locked, they're swapped in and out as needed.

Keep in mind that locking an area essentially reduces the amount of RAM that can be used by other applications, and so increases the likelihood of swapping. So you shouldn't lock simply because you're greedy. But if the area that you're locking is going to be shared among some number of other applications, or if you're writing a real-time application that processes large chunks of data, then locking can be a benefit.

The locking scheme is set by the **create_area()** function and is thereafter immutable. You can't re-declare the lock when you clone an area.

Using an Area

Ultimately, you use an area for the virtual memory that it represents: You create an area because you want some memory to which you can write and from which you can read data. These acts are performed in the usual manner, through references to specific addresses. Setting a pointer to a location within the area, and checking that you haven't exceeded the area's memory bounds as you increment the pointer (while reading or writing) are your own responsibility. To do this properly, you need to know the area's starting address and its extent:

- An area's starting address is maintained as the **address** field in its **area_info** structure; you retrieve the **area_info** for a particular area through the **get_area_info()** function.
- The size of the area (in bytes) is given as the **size** field of its **area_info** structure.

An important point, with regard to **area_info**, is that the **address** field is only valid for the application that created or cloned the area (in other words, the application that created the **area_id** that was passed to **get_area_info()**). Although the memory that underlies an area is global, the address that you get from an **area_info** structure refers to a specific address space.

If there's any question about whether a particular **area_id** is "local" or "foreign," you can compare the **area_info.team** field to your thread's team.

Deleting an Area

When your application quits, the areas (the **area_id** numbers) that it created through **create_area()** or **clone_area()** are automatically rendered invalid. The memory underlying these areas, however, isn't necessarily freed. An area's memory is freed only when (and as soon as) there are no more areas that refer to it.

You can force the invalidation of an **area_id** by passing it to the **delete_area()** function. Again, the underlying memory is only freed if yours is the last area to refer to the memory.

Deleting an area, whether explicitly through **delete_area()**, or because your application quit, never affects the status of other areas that were cloned from it.

Functions

area_for()

> area_id **area_for(**void **addr***)**

Returns the **area_id** of the area that contains the given address within your own team's address space. The argument needn't be the starting address of an area, nor must it start on a page boundary: If the address lies anywhere within one of your application's areas, the ID of that area is returned.

Since the address is taken to be in the local address space, the area that's returned will also be local—it will have been created or cloned by your application.

If the address doesn't lie within an area, **B_ERROR** is returned.

See also: **find_area()**

clone_area()

```
long clone_area(const char *clone_name,
                void **clone_addr,
                ulong clone_addr_spec,
                ulong clone_protection,
                area_id source_area)
```

Creates a new area (the *clone* area) that maps to the same physical memory as an existing area (the *source* area). The arguments are:

- *clone_name* is the name that you wish to assign to the clone area. Area names are, at most, **B_OS_NAME_LENGTH** (32) characters long.
- *clone_addr* points to a value that gives the address at which you want the clone area to start; the pointed-to value must be a multiple of **B_PAGE_SIZE** (4096). The function sets the value pointed to by *clone_addr* to the area's actual starting address—it may be different from the one you requested. The constancy of **clone_addr* depends on the value of *clone_addr_spec*, as explained next.
- *clone_addr_spec* is one of four constants that describes how *clone_addr* is to be interpreted. The first three constants, **B_EXACT_ADDRESS**, **B_BASE_ADDRESS**, and **B_ANY_ADDRESS**, have meanings as explained under **create_area()**.

  The fourth constant, **B_CLONE_ADDRESS**, specifies that the address of the cloned area should be the same as the address of the source area. Cloning the address is convenient if you have two (or more) applications that want to pass pointers to each other—by using cloned addresses, the applications won't have to offset the pointers that they receive. For both the **B_ANY_ADDRESS** and **B_CLONE_ADDRESS** specifications, the value that's pointed to by the *clone_addr* argument is ignored.

- *clone_protection* is one or both of **B_READ_AREA** and **B_WRITE_AREA**. These have the same meaning as in **create_area()**; keep in mind, as described there, that a cloned area can have a protection that's different from that of its source.

- *source_area* is the **area_id** of the area that you wish to clone. You usually supply this value by passing an area name to the **find_area()** function.

The cloned area inherits the source area's locking scheme (**B_FULL_LOCK**, **B_LAZY_LOCK**, or **B_NO_LOCK**).

Usually, the source area and clone area are in two different applications. It's possible to clone an area from a source that's in the same application, but there's not much reason to do so unless you want the areas to have different protections.

If **area_clone()** clone is successful, the clone's **area_id** is returned. Otherwise, the function returns one of the following error constants:

Constant	Meaning
B_BAD_VALUE	Bad argument value; you passed an unrecognized constant for *addr_spec* or *lock*, the *addr* value isn't a multiple of **B_PAGE_SIZE**, you set *addr_spec* to **B_EXACT_ADDRESS** or **B_CLONE_ADDRESS** but the address request couldn't be fulfilled, or *source_area* doesn't identify an existing area.
B_NO_MEMORY	Not enough memory to allocate the system structures that support this area.
B_ERROR	Some other system error prevented the area from being created.

See also: **create_area()**, **delete_area()**

create_area()

area_id **create_area(**const char **name*,
void ***addr*,
ulong *addr_spec*,
ulong *size*,
ulong *lock*,
ulong *protection***)**

Creates a new area and returns its **area_id**. The arguments are:

- *name* is the name that you wish to assign to the area. It needn't be unique. Area names are, at most, **B_OS_NAME_LENGTH** (32) characters long.

- *addr* points to the address at which you want the area to start. The value of **addr* must signify a page boundary; in other words, it must be an integer multiple of **B_PAGE_SIZE** (4096). Note that this is a pointer to a pointer: **addr*—not *addr*—

should be set to the desired address; you then pass the address of *addr* as the argument, as shown below:

```
/* Set the address to a page boundary. */
char *addr = (char *)(4096 * 100);

/* Pass the address of addr as the second argument. */
create_area( "my area", &addr, ...);
```

The function sets the value of **addr* to the area's actual starting address—it may be different from the one you requested. The constancy of **addr* depends on the value of *addr_spec*, as explained next.

- *addr_spec* is a constant that tells the function how the **addr* value should be applied. There are three address specification constants:

  B_EXACT_ADDRESS means you want the value of **addr* to be taken literally and strictly. If the area can't be allocated at that location, the function fails.

  B_BASE_ADDRESS means the area can start at a location equal to or greater than **addr.*

  B_ANY_ADDRESS means the starting address is determined by the system. In this case, the value that's pointed to by *addr* is ignored (going into the function).

  (A fourth specification, **B_CLONE_ADDRESS**, is only used by the **clone_area()** function.)

- *size* is the size, in bytes, of the area. The size must be an integer multiple of **B_PAGE_SIZE** (4096). The upper limit of *size* depends on the available swap space (or RAM, if the area is to be locked).

- *lock* describes how the physical memory should be treated with regard to swapping. There are three locking constants:

  B_FULL_LOCK means the area's memory is immediately locked into RAM and won't be swapped out.

  B_LAZY_LOCK allows individual pages of memory to be brought into RAM through the natural order of things and *then* locks them.

  B_NO_LOCK means pages are never locked, they're swapped in and out as needed.

- *protection* is a mask that describes whether the memory can be written and read. You form the mask by adding the constants **B_READ_AREA** (the area can be read) and **B_WRITE_AREA** (it can be written). The protection you describe applies only to this area. If your area is cloned, the clone can specify a different protection.

If **create_area()** is successful, the new **area_id** number is returned. If it's unsuccessful, one of the following error constants is returned:

Constant	Meaning
B_BAD_VALUE	Bad argument value. You passed an unrecognized constant for *addr_spec* or *lock*, the *addr* or *size* value isn't a multiple of **B_PAGE_SIZE**, or you set *addr_spec* to **B_EXACT_ADDRESS** but the address request couldn't be fulfilled.
B_NO_MEMORY	Not enough memory to allocate the necessary system structures that support this area. Note that this error code *doesn't* mean that you asked for too much physical memory.
B_ERROR	Some other system error prevented the area from being created. Most notably, **B_ERROR** is returned if *size* is too large.

See also: **clone_area()**, **delete_area()**

delete_area()

long **delete_area(**area_id *area***)**

Deletes the designated area. If no one other area maps to the physical memory that this area represents, the memory is freed.

Note: Currently, anybody can delete any area—the act isn't denied if, for example, the **area_id** argument was created by another application. This freedom will be rescinded in a later release. Until then, try to avoid deleting other application's areas.

If *area* doesn't designate an actual area, this function returns **B_ERROR**; otherwise it returns **B_NO_ERROR**.

See also: **create_area()**, **clone_area()**

find_area()

area_id **find_area(**const char **name***)**

Returns an area that has a name that matches the argument. Area names needn't be unique—successive calls to this function with the same argument value may not return the same **area_id**.

What you do with the area you've found depends on where it came from:

- If you're finding an area that your own application created or cloned, you can use the returned ID directly.
- If the area was created or cloned by some other application, you should immediately clone the area (unless you're doing something truly innocuous, such as simply examining the area's info structure).

If the argument doesn't identify an existing area, the **B_NAME_NOT_FOUND** error code is returned.

See also: **area_for()**

get_area_info(), get_nth_area_info()

long **get_area_info(**area_id *area*, area_info **info***)**
long **get_nth_area_info(**team_id *team*, long *n*, area_info **info***)**

Copies information about a particular area into the **area_info** structure designated by *info*. The first version of the function designates the area directly, by **area_id**. The second version designates the *n*'th area within the given team. If the *team* argument is 0, all teams are considered.

The **area_info** structure is defined as:

```
typedef struct area_info {
        area_id  area;
        char  name[B_OS_NAME_LENGTH];
        void  *address;
        ulong  size;
        ulong  lock;
        ulong  protection;
        team_id  team;
        ulong  ram_size;
        ulong  copy_count;
        ulong  in_count;
        ulong  out_count;
} area_info;
```

The fields are:

- **area** is the **area_id** that identifies the area. This will be the same as the function's *area* argument.
- **name** is the name that was assigned to the area when it was created or cloned.
- **address** is a pointer to the area's starting address. Keep in mind that this address is only meaningful to the application that created (or cloned) the area.
- **size** is the size of the area, in bytes.
- **lock** is a constant that represents the area's locking scheme. This will be one of **B_FULL_LOCK**, **B_LAZY_LOCK**, or **B_NO_LOCK**.
- **protection** specifies whether the area's memory can be read or written. It's a combination of **B_READ_AREA** and **B_WRITE_AREA.**
- **team** is the **team_id** of the thread that created or cloned this area.

The final four fields give information about the area that's useful in diagnosing system use. The fields are particularly valuable if you're hunting for memory leaks:

- **ram_size** gives the amount of the area, in bytes, that's currently swapped in.
- **copy_count** is a "copy-on-write" count that can be ignored—it doesn't apply to the areas that you create. The system can create copy-on-write areas (it does so when it loads the data section of an executable, for example), but you can't.
- **in_count** is a count of the total number of times any of the pages in the area have been swapped in.
- **out_count** is a count of the total number of times any of the pages in the area have been swapped out.

If the *area* argument doesn't identify an existing area, the function returns **B_BAD_VALUE**; otherwise it returns **B_NO_ERROR**.

resize_area()

long **resize_area(**area_id *area*, ulong *new_size***)**

Sets the size of the designated area to *new_size*, measured in bytes. The *new_size* argument must be a multiple of **B_PAGE_SIZE** (4096).

Size modifications affect the end of the area's existing memory allocation: If you're increasing the size of the area, the new memory is added to the end of area; if you're shrinking the area, end pages are released and freed. In neither case does the area's starting address change, nor is existing data modified (expect, of course, for data that's lost due to shrinkage).

If the function is successful, **B_NO_ERROR** is returned. Otherwise one of the following error codes is returned:

Constant	Meaning
B_BAD_VALUE	Either *area* doesn't signify a valid area, or *new_size* isn't a multiple of **B_PAGE_SIZE**.
B_NO_MEMORY	Not enough memory to allocate the system structures that support the new portion of the area. This should only happen if you're increasing the size of the area. Note that this error code *doesn't* mean that you asked for too much physical memory.
B_ERROR	Some other system error prevented the area from being created. Most notably, **B_ERROR** is returned if *new_size* is too large.

See also: **create_area()**

set_area_protection()

long **set_area_protection(**area_id *area*, ulong *new_protection***)**

Sets the given area's read and write protection. The *new_protection* argument is a mask that specifies one or both of the values **B_READ_AREA** and **B_WRITE_AREA**. The former means that the area can be read; the latter, that it can be written to. An area's protection only applies to access to the underlying memory through that specific area. Different area clones that refer to the same memory may have different protections.

The function fails (the old protection isn't changed) and returns **B_BAD_VALUE** if area doesn't identify a valid area; otherwise it returns **B_NO_ERROR**.

See also: **create_area()**

Images

Declared in: <kernel/image.h>

Overview

An *image* is compiled code; put another way, an image is what the compiler produces. There are three types of images:

- An *app image* is an application. Every application has a single app image.
- A *library image* is a dynamically linked library (a "shared library"). Most applications link against the system library (**libbe.so**) that Be provides.
- An *add-on image* is an image that you load into your application as it's running. Symbols from the add-on image are linked and references are resolved when the image is loaded. Thus, an add-on image provides a sort of "heightened dynamic linking" beyond that of a DLL.

The following sections explain how to load and run an app image, how to create a shared library, and how to create and load an add-on image.

Loading an App Image

Loading an app image is like running a "sub-program." The image that you load is launched in much the same way as had you double-clicked it in the Browser, or launched it from the command line. It runs in its own team—it doesn't share the address space of the application from which it was launched—and, generally, leads its own life.

Any application can be loaded as an app image; you don't need to issue special compile instructions or otherwise manipulate the binary. The one requirement of an app image is that it must have a **main()** function; hardly a restrictive request.

To load an app image, you call the **load_executable()** function, the protocol for which is:

```
thread_id load_executable(BFile *file,
                          int argc,
                          const char **argv,
                          const char **env)
```

The function takes, as its first argument, a BFile object that represents the image file. Having located the file, the function creates a new team, spawns a main thread in that team, and then returns the **thread_id** of that thread to you. The thread that's returned is the executable's main thread. It won't be running: To make it run you pass the **thread_id** to **resume_thread()** or **wait_for_thread()** (as explained in the major section "Threads and Teams").

In addition to the BFile argument, **load_executable()** takes an *argc*/*argv* argument pair (which are copied and forwarded to the new thread's **main()** function), as well as a pointer to an array of environment variables (strings):

- The *argc*/*argv* arguments must be set up properly—you can't just pass 0 and **NULL**. To properly instantiate the arguments, the first string in the *argv* array must be the name of the image file (in other words, the name of the program that you're going to launch). You then install any other arguments you want in the array, and terminate the array with a **NULL** entry. *argc* is set to the number of entries in the *argv* array (not counting the terminating **NULL**). It's the caller's responsibility to free the *argv* array after **load_executable()** returns.

- *envp* is an array of environment variables that are also passed to **main()**. Typically, you use the global **environ** pointer (which you must declare as an **extern**—see the example, below). You can, of course, create your environment variable array: As with the *argv* array, the *envp* array should be terminated with a **NULL** entry, and you must free the array when **load_executable()** returns (that is, if you allocated it yourself—don't try to free **environ**).

The following example demonstrates a typical use of **load_executable()**. First, we include the appropriate files and declare the necessary variables:

```
#include <image.h>  /* load_executable() */
#include <OS.h>     /* wait_for_thread() */
#include <stdlib.h> /* malloc() */

/* Here's how you declare the environment variable array. */
extern char **environ;

BFile exec_file;
record_ref exec_ref;
char **arg_v; /* choose a name that doesn't collide with argv */
long arg_c; /* same here vis a vis arg_c */
thread_id exec_thread;
long return_value;
```

Next, we set our BFile's ref so the object refers to the executable file, which we're calling **adder**. For this example, we use **get_ref_for_path()** to set the ref's value (see the Storage Kit chapter for more information on these manipulations):

```
get_ref_for_path("/hd/my_apps/adder", &exec_ref);
exec_file.SetRef(exec_ref);
```

Install, in the **arg_v** array, the "command line" arguments that we're sending to **adder**. Let's pretend the **adder** program takes two integers, adds them together, and returns the result as **main()**'s exit code. Thus, there are three arguments: The name of the program ("adder"), and the values of the two addends converted to strings. Since there are three arguments, we allocate **arg_v** to hold four pointers (to accommodate the final **NULL**). Then we allocate and copy the arguments.

```
arg_c = 3;
arg_v = (char **)malloc(sizeof(char *) * (agc + 1));

arg_v[0] = strdup("adder");
arg_v[1] = strdup("5");
arg_v[2] = strdup("3");
arg_v[3] = NULL;
```

Now that everything is properly set up, we call **load_executable()**. After the function returns, it's safe to free the allocated **arg_v** array:

```
exec_thread=load_executable(&exec_file, arg_c, arg_v, environ);
free(arg_v);
```

At this point, **exec_thread** is suspended (the natural state of a newly-spawned thread). In order to retrieve its return value, we use **wait_for_thread()** to tell the thread to run:

```
wait_for_thread(exec_thread, &return_value);
```

After **wait_for_thread()** returns, the value of **return_value** should be 8 (i.e. 5 + 3).

Creating a Shared Library

The primary documentation for creating a shared library is provided by MetroWerks in their CodeWarrior manual. Beyond the information that you find there, you should be aware of the following amendments and caveats.

- You mustn't export your library's symbols through the **-export all** compiler flag. Instead, you should either use **-export pragma** or **-@export *filename*** (which is the same as **-f *filename***). See the MetroWerks manual for details on how to use these flags.

- The libraries that you create must be placed in **/system/lib** so the loader can find them when an application (that's uses your libraries) is launched.

Creating and Using an Add-on Image

An add-on image is indistinguishable from a shared library image. Creating an add-on is, therefore, exactly like creating a shared library, a topic that we breezed through immediately above. The one exception to the rules given above is in where the add-on must live: You can keep your add-ons anywhere in the file system. When you load an add-on (through the **load_add_on()** function), you have to refer to the add-on file directly through the use of a BFile—the system doesn't search for the file for you.

Loading an Add-on Image

To load an add-on into your application, you call the **load_add_on()** function. The function takes a pointer to a BFile object that refers to the add-on file, and returns an **image_id** number that uniquely identifies the image across the entire system.

For example, let's say you've created an add-on image that's stored in the file **/hd/addons/adder** (the add-on will perform the same adding operation that was demonstrated in the **load_executable()** example). The code that loads the add-on would look like this:

```
/* For brevity, we won't check errors.  */
BFile addon_file;
record_ref addon_ref;
image_id addon_image;

/* Establish the file's ref.  */
get_ref_for_path("/hd/addons/adder", &addon_ref);
addon_file.SetRef(addon_ref);

/* Load the add-on. */
addon_image = load_add_on(&addon_file);
```

Unlike loading an executable, loading an add-on doesn't create a separate team (nor does it spawn another thread). The whole point of loading an add-on is to bring the image into your application's address space so you can call the functions and fiddle with the variables that the add-on defines.

Symbols

After you've loaded an add-on into your application, you'll want to examine the symbols (variables and functions) that it has brought with it. To get information about a symbol, you call the **get_image_symbol()** function:

long **get_image_symbol(**image_id *image*,
char **symbol_name*,
long *symbol_type*,
void ***location***)**

The function's first three arguments identify the symbol that you want to get:

- The first argument is the **image_id** of the add-on that owns the symbol.
- The second argument is the symbol's name. This assumes, of course, that you know the name. In general, using an add-on implies just this sort of cooperation.
- The third is a constant that gives the symbol's *symbol type*. The only types you should care about are **B_SYMBOL_TYPE_DATA** which you use for variables, and **B_SYMBOL_TYPE_TEXT** which you use for functions.

The function returns, by reference in its final argument, a pointer to the symbol's address. For example, let's say the **adder** add-on code looks like this:

```
long addend1 = 0;
long addend2 = 0;

long adder(void)
{
    return (addend1 + addend2);
}
```

To examine the variables (**addend1** and **addend2**), you would call **get_image_symbol()** thus:

```
long *var_a1, *var_a2;

/* addon_image is the image_id that was returned by the
 * load_add_on() call in the previous example.
 */
get_image_symbol(addon_image, "addend1", 1, &var_a1);
get_image_symbol(addon_image, "addend2", 1, &var_a2);
```

To get the symbol for the **adder()** function is a bit more complicated. The compiler renames a function's symbol to encode the data types of the function's arguments. The encoding scheme is explained in the next section; to continue with the example, we'll simply accept that the **adder()** function's symbol is

```
adder__Fv
```

And so...

```
long (*func_add)();
get_image_symbol(addon_image, "adder__Fv", 2, &func_add);
```

Now that we've retrieved all the symbols, we can set the values of the two addends and call the function:

```
*var_a1 = 5;
*var_a2 = 3;
long return_value = (*func_add)();
```

Function Symbol Encoding

The compiler encodes function symbols according to this format:

*functionName*__**F**<*arg1Type*><*arg2Type*><*arg3Type*>....

where the argument type codes are

Code	Type
i	**int**
l	**long**

f	**float**
d	**double**
c	**char**
v	**void**

In addition, if the argument is declared as unsigned, the type code character is preceded by "U". If it's a pointer, the type code (and, potentially, the "U") is preceded by "P"; a pointer to a pointer is preceded by "PP". For example, a function that's declared as

```
void Func(long, unsigned char **, float *, double);
```

would have the following symbol name:

```
Func__FlUPPcPfd
```

Note that **typedef**'s are translated to their natural types. So, for example, this:

```
void dump_thread(thread_id, bool);
```

becomes

```
dump_thread__FlUc
```

Functions

get_image_info(), get_nth_image_info()

long **get_image_info(**image_id *image*, image_info **info***)**

long **get_nth_image_info(**team_id *team*,
long *n*,
image_info **info***)**

These functions return information about a particular image. The first version identifies the image by its first argument; the second version locates the *n*'th image that's loaded into *team*. The information is returned in the *info* argument. The **image_info** structure is defined as:

```
typedef struct {
      long  volume;
      long  directory;
      char  name[B_FILE_NAME_LENGTH];
      image_id id;
      void  *text;
      long  text_size;
      void  *data;
      long  data_size;
      image_type type;
} image_info
```

The volume and directory fields are, practically speaking, private. The other fields are:

- **name**. The name of the file whence sprang the image.
- **id.** The image's **image_id** number.
- **text** and **text_size**. The address and the size (in bytes) of the image's text segment.
- **data** and **data_size**. The address and size of the image's data segment.
- **type**. A constant that tells whether this is an app, library, or add-on image.

The self-explanatory **image_type** constants are:

- **B_APP_IMAGE**
- **B_LIBRARY_IMAGE**
- **B_ADD_ON_IMAGE**

The functions return **B_BAD_IMAGE_ID** or **B_BAD_INDEX** if the designated image doesn't exist. Otherwise, they return **B_NO_ERROR**.

get_image_symbol(), get_nth_image_symbol()

long **get_image_symbol(**image_id *image*,
char *symbol_name*,
long *symbol_type*,
void ***location***)**

long **get_nth_image_symbol(**image_id *image*,
long *n*,
char **name*,
int **name_length*,
int **symbol_type*,
void ***location***)**

get_image_symbol() returns, in *location*, a pointer to the address of the symbol that's identified by the *image*, *symbol_name*, and *symbol_type* arguments. An example demonstrating the use of this function is given in "Symbols" on page 58.

get_nth_image_symbol() returns information about the *n*'th symbol in the given image. The information is returned in the arguments:

- *name* is the name of the symbol. You have to allocate the *name* buffer before you pass it in—the function copies the name into the buffer.

- You point *name_length* to an integer that gives the length of the *name* buffer that you're passing in. The function uses this value to truncate the string that it copies into *name*. The function then resets *name_length* to the full (untruncated) length of the symbol's name (plus one byte to accommodate a terminating **NULL**). To ensure that you've gotten the symbol's full name, you should compare the in-going value of *name_length* with the value that the function sets it to. If the in-going value is less than the full length, you can then re-invoke **get_nth_image_symbol()** with an adequately lengthened *name* buffer, and an increased *name_length* value.

Important: Keep in mind that *name_length* is reset each time you call **get_nth_image_symbol()**. If you're calling the function iteratively (to retrieve all the symbols in an image), you need to reset the *name_length* value between calls.

- The function sets *symbol_type* to **B_SYMBOL_TYPE_DATA** if the symbol is a variable, or **B_SYMBOL_TYPE_TEXT** if the symbol is a function. The argument's value going into the function is of no consequence.

- The function sets *location* to point to the symbol's address.

To retrieve **image_id** numbers on which these functions can act, use the **get_nth_image_info()** function. Such numbers are also returned directly when you load an add-on image through the **load_add_on()** function.

The functions return **B_BAD_IMAGE_ID** or **B_BAD_INDEX** if the designated image doesn't exist. Otherwise, they return **B_NO_ERROR**.

load_add_on(), unload_add_on()

image_id **load_add_on(**BFile **file***)**
long **unload_add_on(**image_id *image***)**

load_add_on() loads an add-on image, identified by *file*, into your application's address space. The function returns an **image_id** (a positive integer) that represents the loaded image. An example that demonstrates the use of **load_add_on()** is given in "Loading an Add-on Image" on page 58.

You can load the same add-on image twice; each time you load the add-on a new, unique **image_id** is created and returned. If the requested file couldn't be loaded as an add-on (for whatever reason), the function returns **B_ERROR**.

unload_add_on() removes the add-on image identified by the argument. The image's symbols are removed, and the memory that they represent is freed. If the argument doesn't identify a valid image, the function returns **B_ERROR**. Otherwise, it returns **B_NO_ERROR**.

load_executable()

```
thread_id load_executable(BFile *file,
                          int argc,
                          const char **argv,
                          const char **env)
```

Loads an app image into the system (it *doesn't* load the image into the caller's address space), creates a separate team for the new application, and spawns and returns the ID of the team's main thread. The image is identified by the *file* argument; *file* must have its ref set before this function is called. It's of no consequence whether the object is open or closed when you call this function.

The other arguments are passed to the image's **main()** function (they show up there as the function's similarly named arguments):

- *argc* gives the number of entries that are in the *argv* array.
- The first string in the *argv* array must be the name of the image file (in other words, the name of the program that you're going to launch). You then install any other arguments you want in the array, and terminate the array with a **NULL** entry. Note that the value of *argc* shouldn't count *argv*'s terminating **NULL**.
- *envp* is an array of environment variables that are also passed to **main()**. Typically, you use the global **environ** pointer:

```
extern char **environ;

load_executable(..., environ);
```

The *argv* and *envp* arrays are copied into the new thread's address space. If you allocated either of these arrays, it's safe to free them immediately after **load_executable()** returns.

The thread that's returned by **load_executable()** is in a suspended state. To start the thread running, you pass the **thread_id** to **resume_thread()** or **wait_for_thread()**.

An example that demonstrates the use of **load_executable()** is given in "Loading an App Image" on page 55.

If the function returns **B_ERROR** upon failure.

Miscellaneous Functions, Constants, and Defined Types

Miscellaneous Functions

debugger()

void **debugger(**const char **string***)**

Throws the calling thread into the debugger. The *string* argument becomes the debugger's first utterance.

get_system_info()

long **get_system_info(**system_info **info***)**

Returns information about the computer. The information is returned in *info*, a **system_info** structure.

is_computer_on()

long **is_computer_on(**void**)**

Returns 1 if the computer is on. If the computer isn't on, the value returned by this function is undefined.

system_time()

double **system_time(**void**)**

Returns the number of microseconds that have elapsed since the computer was booted.

Constants

Area Location Constants

<kernel/OS.h>

Constant	Meaning
B_ANY_ADDRESS	Put the area anywhere
B_EXACT_ADDRESS	The area must start exactly at a given address
B_BASE_ADDRESS	The area can start anywhere above a given address
B_CLONE_ADDRESS	The clone must start at the same address as the original

These constants represent the different locations at which an area can be placed when it's created. They're used as values for the address arguments in **create_area()** and **clone_area()**. **B_CLONE_ADDRESS** can be passed to **clone_area()** only; the other three can be passed to either function.

See also: "Areas" on page 45

Area Lock Constants

<kernel/OS.h>

Constant	Meaning
B_NO_LOCK	Never lock the area's pages
B_LAZY_LOCK	Lock pages as they're swapped in
B_FULL_LOCK	Lock all pages now

These constants represent an area's "locking scheme," the circumstances in which the area's underlying memory is locked into RAM. You set the locking scheme for an area by passing one of these constants to **create_area()**'s lock argument; the scheme can't be changed thereafter.

To query an area's locking scheme, retrieve its **area_info** structure (through **get_area_info()**) and look at the **lock** field.

See also: "Areas" on page 45

Area Protection Constants

<kernel/OS.h>

Constant	Meaning
B_READ_AREA	The area can be read from
B_WRITE_AREA	The area can be written into

These constants represent the read and write protection that's enforced for an area. The constants are flags that can be added together and passed as the protection argument to

create_area() and **clone_area()**. You can change an area's protection through the **set_area_protection()** function.

To query an area's protection, retrieve its **area_info** structure (through **get_area_info()**) and look at the **protection** field.

See also: "Areas" on page 45

CPU Count

<kernel/OS.h>

Constant	Value
B_MAX_CPU_NUM	8

This constant gives the maximum number of CPUs that a single system can support. The **cpu_count** field of the **system_info** structure gives the number of CPUs that are actually on a given system. To retrieve this structure, call the **get_system_info()** function.

See also: **get_system_info()**

CPU Type Constant

<kernel/OS.h>

Constant
B_CPU_PPC_601
B_CPU_PPC_603
B_CPU_PPC_603e
B_CPU_PPC_604
B_CPU_PPC_604e
B_CPU_PPC_686

These constants represent the different CPU chips that the BeBox has used/is using/might use. To discover which chip the local machine is using, looking in the **cpu_type** field of the **system_info** structure. To retrieve this structure, call the **get_system_info()** function.

See also: **get_system_info()**

File Name Length

<kernel/OS.h>

Constant	Value
B_FILE_NAME_LENGTH	64

This constant gives the maximum length of the name of a file or directory.

Image Type Constants

<kernel/image.h>

Constant	Meaning
B_APP_IMAGE	Application image
B_LIBRARY_IMAGE	Shared library image
B_ADD_ON_IMAGE	Add-on image
B_SYSTEM_IMAGE	System-defined image

The image type constants (type **image_type**) enumerate the different *images*, or loadable compiled code, that you can create or otherwise find on the system. Of the four image types, you can't create **B_SYSTEM_IMAGES**; however, it's possible to run across a system-defined image if you retrieve all the image info structures for all teams (through the **get_nth_image_info()** function).

See also: "Images" on page 55

Image Symbol Type Constants

<kernel/OS.h>

Constant	Meaning
B_SYMBOL_TYPE_DATA	The symbol is a variable
B_SYMBOL_TYPE_TEXT	The symbol is a function

The image symbol type constants describe the nature of a particular image symbol. You retrieve symbol information from an image through the **get_image_symbol()**.

See also: "Images" on page 55

Operating System Doodad Name Length

<kernel/OS.h>

Constant	Value
B_OS_NAME_LENGTH	32

This constant gives the maximum length of the name of a thread, semaphore, port, area, or other operating system bauble.

Page Size

<kernel/OS.h>

Constant	Value
B_PAGE_SIZE	4096

The **B_PAGE_SIZE** constant gives the size, in bytes, of a page of RAM.

Port Message Count

<kernel/OS.h>

Constant	Value
B_MAX_PORT_COUNT	128

This constant gives the maximum number of messages that a port can hold at a time. This value isn't applied automatically—you declare a port's message capacity when you create it.

To query a port's message capacity, retrieve its **port_info** structure (through the **get_port_info()** function) and look at the **capacity** field.

See also: "Ports" on page 23

Semaphore Control Flags

<kernel/OS.h>

Acquire Flag	Meaning
B_TIMEOUT	Honor **acquire_sem_etc()**'s *timeout* argument.
B_CAN_INTERRUPT	The semaphore can be interrupted by a signal.
B_CHECK_PERMISSION	Make sure this isn't a system semaphore.

Release Flag	Meaning
B_DO_NOT_RESCHEDULE	Don't reschedule after the semaphore is released.

These are the flag values that can be passed to the **acquire_sem_etc()** and **release_sem_etc()** functions.

The timeout flag (**B_TIMEOUT**) applies to all semaphore acquisitions: If, having set the **B_TIMEOUT** flag, your **acquire_sem_etc()** call blocks, the acquisition attempt will give up after some number of microseconds (as given in the function's *timeout* argument). If **B_TIMEOUT** isn't set, the acquisition can, potentially, block forever. The other two acquisition flags are used by device driver writers only. The meanings of these flags is given in the Device Kit chapter.

The **B_DO_NOT_RESCHEDULE** flag applies to the **release_sem_etc()** function. Normally, when a semaphore is released, the kernel immediately finds another thread to run, even if the releasing thread hasn't used up a full "schedule quantum" worth of CPU attention. By setting the **B_DO_NOT_RESCHEDULE** flag, you tell the scheduler to let the releasing thread run for its normally alloted amount of time.

Thread Priority Constants

<kernel/OS.h>

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORITY	20

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	100
B_URGENT_PRIORITY	110
B_REAL_TIME_PRIORITY	120

These constants represent the thread priority levels. The higher a thread's priority value, the more attention it gets from the CPUs; the constants are listed here from lowest to highest priority.

There are two priority categories:

- Time-sharing priorities (priority values from 1 to 99).
- Real-time priorities (100 and greater).

A time-sharing thread (a thread with a time-sharing priority value) is executed only if there are no real-time threads in the ready queue. In the absence of real-time threads, a time-sharing thread is elected to run once every "scheduler quantum" (currently, every three milliseconds). The higher the time-sharing thread's priority value, the greater the chance that it will be the next thread to run.

A real-time thread is executed as soon as it's ready. If more than one real-time thread is ready at the same time, the thread with the highet priority is executed first. The thread is allowed to run without being preempted (except by a real-time thread with a higher priority) until it blocks, snoozes, is suspended, or otherwise gives up its plea for attention.

You set a thread's priority when you spawn it (**spawn_thread()**); it can be changed thereafter through the **set_thread_priority()** function. Although you can set a thread priority to values other than those defined by the constants shown here, it's strongly suggested that you stick with the constants.

To query a thread's priority, look at the **priority** field of the thread's **thread_info** structure (which you can retrieve through **get_thread_info()**).

See also: "Threads and Teams" on page 5

Thread State Constants

<kernel/OS.h>

Constant	Meaning
B_THREAD_RUNNING	The thread is currently receiving attention from a CPU.
B_THREAD_READY	The thread is waiting for its turn to run.
B_THREAD_RECEIVING	The thread is sitting in a **receive_data()** call.
B_THREAD_ASLEEP	The thread is sitting in a **snooze()** call.
B_THREAD_SUSPENDED	The thread has been suspended or is freshly-spawned.
B_THREAD_WAITING	The thread is waiting to acquire a semaphore.

These constants (type **thread_state**) represent the various states that a thread can be in. You can't set a thread's state directly; the state changes as the result of the thread's sequence of operations.

You can query a thread's state by looking at the **state** field of its **thread_info** structure. To retrieve the structure, call **get_thread_info()**. Be aware, however, that a thread's state is extremely ephemeral; by the time you retrieve it, it may have changed.

See also: "Threads and Teams" on page 5

System Team ID

<kernel/OS.h>

Constant	Meaning
B_SYSTEM_TEAM	The **team_id** of the kernel's team

The **B_SYSTEM_TEAM** constant identifies the kernel's team. You should only need to use this constant if you're bequeathing ownership of a port or semaphore to the kernel, an activity that's typically the province of driver writers.

Defined Types

area_id

<kernel/OS.h>

typedef long **area_id**

The **area_id** type uniquely identifies area.

See also: "Areas" on page 45

area_info

<kernel/OS.h>

```
typedef struct {
      area_id area;
      char name[B_OS_NAME_LENGTH];
      void *address;
      ulong size;
      ulong lock;
      ulong protection;
      team_id team;
      ulong  ram_size;
      ulong  copy_count;
      ulong  in_count;
      ulong  out_count;
} area_info
```

The **area_info** structure holds information about a particular area. **area_info** structures are retrieved through the **get_area_info()** function. The structure's fields are:

- **area** is the **area_id** that identifies the area.

- **name** is the name that was assigned to the area when it was created or cloned.

- **address** is a pointer to the area's starting address. Keep in mind that this address is only meaningful to the application that created (or cloned) the area.
- **size** is the size of the area, in bytes.
- **lock** is a constant that represents the area's locking scheme. This will be one of **B_FULL_LOCK**, **B_LAZY_LOCK**, or **B_NO_LOCK**.
- **protection** specifies whether the area's memory can be read or written. It's a combination of **B_READ_AREA** and **B_WRITE_AREA.**
- **team** is the **team_id** of the thread that created or cloned this area.

Three of the final four fields (you can ignore the **copy_count** field) give information about the area that's useful in diagnosing system use:

- **ram_size** gives the amount of the area, in bytes, that's currently swapped in.
- **in_count** is the number of times the system has swapped in a page from the area.
- **out_count** is the number of times pages have been swapped out.

See also: "Areas" on page 45

cpu_info

```
<kernel/OS.h>

typedef struct {
        double  active_time;
} cpu_info
```

The **cpu_info** structure describes facets of a particular CPU. Currently, the structure contains only one field, **active_time**, that measures the amount of time, in microseconds, that the CPU has actively been working since the machine was last booted. One structure for each CPU is created and maintained by the system. An array of all such structures can be found in the **cpu_infos** field of the **system_info** structure. To retrieve a **system_info** structure, you call the **get_system_info()** function.

See also: **system_info**

image_info

<kernel/image.h>

```
typedef struct {
        long  volume;
        long  directory;
        char  name[B_FILE_NAME_LENGTH];
        image_id id;
        void  *text;
        long  text_size;
        void  *data;
        long  data_size;
        image_type type;
} image_info
```

The **image_info** structure contains information about a specific image. The fields are:

- The **volume** and **directory** fields are, practically speaking, private.
- **name**. The name of the file whence sprang the image.
- **id.** The image's **image_id** number.
- **text** and **text_size**. The address and the size (in bytes) of the image's text segment.
- **data** and **data_size**. The address and size of the image's data segment.
- **type**. A constant that tells whether this is an app, library, or add-on image.

The self-explanatory **image_type** constants are:

- **B_APP_IMAGE**
- **B_LIBRARY_IMAGE**
- **B_ADD_ON_IMAGE**

image_type

<kernel/image.h>

typedef enum { ... } **image_type**

The **image_type** type defines the different image types.

See also: **Image Type Constants**

machine_id

<kernel/OS.h>

typedef long **machine_id**[2]

The **machine_id** type encodes a 64-bit number that uniquely identifies a particular BeBox. To discover the machine id of the local machine, look in the **id** field of the **system_info** structure. To retrieve this structure, call the **get_system_info()** function.

See also: **get_system_info()**

port_id

<kernel/OS.h>

typedef long **port_id**

The **port_id** type uniquely identifies ports.

See also: "Ports" on page 23

port_info

<kernel/OS.h>

```
typedef struct port_info {
        port_id port;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        long capacity;
        long queue_count;
        long total_count;
} port_info
```

The **port_info** structure holds information about a particular port. It's fields are:

- **port**. The **port_id** number of the port.
- **team**. The **team_id** of the port's team.
- **name**. The name assigned to the port.
- **capacity**. The length of the port's message queue.
- **queue_count**. The number of messages currently in the queue.
- **total_count**. The total number of message that have been read from the port.

Note that the **total_count** number doesn't include the messages that are currently in the queue.

You retrieve a **port_info** structure through the **get_port_info()** function.

See also: "Ports" on page 23

sem_id

<kernel/OS.h>

typedef long **sem_id**

The **sem_id** type uniquely identifies semaphores.

See also: “Semaphores” on page 31

sem_info

<kernel/OS.h>

```
typedef struct sem_info {
        sem_id sem;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        long count;
        thread_id latest_holder;
} sem_info
```

The **sem_info** structure holds information about a given semaphore. The structure’s fields are:

- **sem**. The **sem_id** number of the semaphore.
- **team**. The **team_id** of the semaphore’s owner.
- **name**. The name assigned to the semaphore.
- **count**. The semaphore’s thread count.
- **latest_holder**. The thread that most recently acquired the semaphore.

Note that the thread that’s identified in the **lastest_holder** field may no longer be holding the semaphore—it may have since released the semaphore. The latest holder is simply the last thread to have called **acquire_sem()** (of whatever flavor) on this semaphore.

You retrieve a sem_info structure through the **get_sem_info()** function.

See also: “Semaphores” on page 31

system_info

<kernel/OS.h>

```
typedef struct {
      machine_id id;
      double boot_time;
      long cpu_count;
      long cpu_type;
      long cpu_revision;
      cpu_info cpu_infos[B_MAX_CPU_NUM];
      double cpu_clock_speed;
      double bus_clock_speed;
      long max_pages;
      long used_pages;
      long page_faults;
      long max_sems;
      long used_sems;
      long max_ports;
      long used_ports;
      long max_threads;
      long used_threads;
      long max_teams;
      long used_teams;
      long volume;
      long directory;
      char name [B_FILE_NAME_LENGTH];
} system_info
```

The **system_info** structure holds information about the machine and the state of the kernel. The structure's fields are:

- **id.** The 64-bit number (encoded as two **long**s) that uniquely identifies this machine.

- **boot_time**. The time at which the computer was last booted, measured in microseconds since January 1st, 1970.

- **cpu_count**. The number of CPUs.
- **cpu_type** and **cpu_revision.** The type constant and revision number of the CPUs.
- **cpu_infos**. An array of **cpu_info** structures, one for each CPU.
- **cpu_clock_speed**. The speed (in Hz) at which the CPUs operate.
- **bus_clock_speed**. The speed (in Hz) at which the bus operates.

- **max_***resource***s** and **used_***resource***s**. The five pairs of **max**/**used** fields give the total number of RAM pages, semaphores, and so on, that the system can create, and the number that are currently in use.

- **page_faults**. The number of times the system a read a page of memory into RAM due to a page fault.

- **volume**. The volume (listed by its ID) that contains the kernel.
- **directory**. The directory (listed by its ID) that contains the kernel.
- **name.** The file name of the kernel.

The **directory** field is unusable: Directory ID numbers aren't visible through the present (public) means of file system access. But you can save the directory IDs that you collect now and trade them in for a higher draft pick next season.

You retrieve a **system_info** structure through the **get_system_info()** function.

See also: **get_system_info()**

team_id

<kernel/OS.h>

typedef long **team_id**

The **team_id** type uniquely identifies teams.

See also: "Threads and Teams" on page 5

team_info

<kernel/OS.h>

```
typedef struct {
        team_id team;
        long thread_count;
        long image_count;
        long area_count;
        thread_id debugger_nub_thread;
        port_id debugger_nub_port;
        long argc;
        char args[64];
} team_info
```

The **team_info** structure holds information about a team. It's returned by the **get_team_info()** function. It's fields are:

- **team** is the team's ID number.

- **thread_count**, **image_count**, and **area_count** give the number of threads that have been spawned, images that have been loaded, and areas that have been created or cloned within this team.

- **debugger_nub_thread** and **debugger_nub_port** are used to communicate with the debugger. Unless you're designing your own debugger, you can ignore these fields.

- The **argc** field is the number of command line arguments that were used to launch the team; **args** is a copy of the first 64 characters from the command line invocation. If this team is an application that was launched through the user interface (by double-clicking, or by accepting a dropped icon), then **argc** is 1 and **args** is the name of the application's executable file.

See also: "Threads and Teams" on page 5

thread_entry

<kernel/OS.h>

typedef long (***thread_entry**)(void *)

The **thread_entry** type is a function protocol for functions that are used as the entry points for new threads. You assign an entry function to a thread when you all **spawn_thread()**; the function takes a **thread_entry** function as its first argument.

See also: "Threads and Teams" on page 5

thread_id

<kernel/OS.h>

typedef long **thread_id**

The **thread_id** type uniquely identifies threads.

See also: "Threads and Teams" on page 5

thread_info

<kernel/OS.h>

```
typedef struct {
        thread_id thread;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        thread_state state;
        long priority;
        sem_id sem;
        double time;
        void *stack_base;
        void *stack_end;
} thread_info
```

This structure holds information about a thread. It's returned by functions such as **get_thread_info()**. The fields are:

- **thread**. The **thread_id** number of the thread.
- **team**. The **team_id** of the thread's team.
- **name**. The name assigned to the thread.
- **state**. A constant that describes what the thread is currently doing.
- **priority**. A constant that represents the level of attention the thread gets.
- **sem**. If the thread is waiting to acquire a semaphore, this is the **sem_id** number of that semaphore. The **sem** field is only valid if the thread's state is **B_THREAD_WAITING**.
- **time**. The amount of active attention the thread has received from the CPUs, measured in microseconds.
- **stack_base**. A pointer to the first byte of memory in the thread's execution stack.
- **stack_end**. A pointer to the last byte of memory in the thread's execution stack.

See also: "Threads and Teams" on page 5

thread_state

<kernel/OS.h>

typedef enum { ... } **thread_state**

The **thread_state** type represents values that describe the various states that a thread can be in.

See also: **Thread State Constants**