The Kernel Kit

The Device Kit − Table of Contents

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 three sections...

- "System Information",
- "Time Information",
- and "Miscellaneous Functions and Constants"

...fill in the gaps.

Areas Concepts

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 features that an area provides that you *don't* get with "standard" memory are these:

- **Areas can be shared**. 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.
- **Areas can be locked into RAM**. 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 can be read− and write−protected**.
- **Areas are page−aligned**. 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.

Because areas are largeone page, minimumyou 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.

In all particulars (but one) you treat the memory that an area gives you exactly as you would treat any allocated memory: You can read and write it through pointer manipulation, or through standard functions such as **memcpy()** and **strcpy()**. The one difference is between areas and malloc'd memory is...

- You never **free()** the memory that an area allocates for you. If you want to get rid of an area, use the **delete_area()** function, instead.

Area IDs and Area Names

Each area that you create is tagged with an **area_id** number:

- An **area_id** number is a positive integer that's global and unique within the scope of the computer. They're not unique across the network, nor are they persistent across boots.
- The **area_id** numbers are generated and assigned automatically by the **create_area()** and **clone_area()** functions. The other area functions operate on these **area_id** numbers (they're required as arguments).
- Although they are global, **area_id** numbers have little meaning outside of the address space (application) in which they were created.
- Once assigned, the **area_id** number doesn't change; the number is invalidated when **delete_area()** is called or when the application (team) that created it dies.
- Don't worry about recycled **area_id** numbers. When an area is deleted, it's **area_id** goes with it. (**area_id** values *are* recycled, but the turnover is at 2^31.)

Areas can also be (loosely) identified by name:

- When you create an area (through **create_area()** or **clone_area()**), you get to name it.
- Area names are not uniqueany number of areas can be assigned the same name.
- To look up an area by name, use the **FindArea()** function.

Sharing an Area Between Applications

For multiple applications to share a common area, one of the applications has to *create* the area, and the other applications *clone* the area. You clone an area by calling **clone_area()**. The function takes, as its last argument, the **area_id** of the source area and returns a new (unique) **area_id** number. All further references to the cloned area (in the cloning application) must be based on the area_id that's returned by **clone_area()**.

So how does a cloner find a source **area_id** in the first place?

- The source application can pass the "original" **area_id** number to the cloners.
- The cloners can find the area by name, by calling **find_area()**.

Keep in mind that area names are not forced to be unique, so the **find_area()** method has some amount of uncertainty. But this can be minimized through clever name creation.

Cloned Memory

The physical memory that lies beneath an area is never implicitly copiedfor example, the area mechanism doesn't perform a "copy−on−write." If two areas refer to the same memory because of cloning, a data modification that's affected through one area will be seen by the other area.

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	The area's memory is locked into RAM when the area is created, and won't be swapped out.
B_CONTIGUOUS	Not only is the area's memory locked into RAM, it's also guaranteed to be contiguous. This is particularlyand perhaps exclusivelyuseful to designers of certain types of device drivers.
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	Pages are never locked, they're swapped in and out as needed.
B_LOMEM	This is a special constant that's used for for areas that need to be locked, contiguous, and that fit within the first 16MB of physical memory. The folks that need this constant know who they are.

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 justifiable excess.

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.

Area Info

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.

Areas

Declared in: be/kernel/OS.h

Library: libroot.so

An area is a chunk of virtual memory that can be shared between threads (possibly in different teams). If your application needs to allocate large chunks of memory, or wants to share lots of data with another application, you should consider using an area.

For more on area concepts, see "Areas Concepts".

For examples of creating and sharing areas, see "Area Examples".

Area Functions

area_for()

```
area_id area_for(void *addr)
```

Returns 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 localit will have been created or cloned by your application.

RETURN CODES

- **B_ERROR**. The address doesn't lie within an area.

See also: find_area()

clone_area()

```
area_id clone_area(const char *clone_name,
    void **clone_addr,
    uint32 clone_addr_spec,
    uint32 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).

- *clone_name* is the name that you wish to assign to the clone area. Area names are, at most, **B_OS_NAME_LENGTH** 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 addressit 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 otherby 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.

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 **clone_area()** clone is successful, the clone's **area_id** is returned. Otherwise, it returns a descriptive error code, listed below.

RETURN CODES

- **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 (unlikely).
- **B_ERROR**. Some other system error prevented the area from being created.

See also: `create_area()`

---

create_area()

```
area_id create_area(const char *name,
    void **addr,
    uint32 addr_spec,
    uint32 size,
    uint32 lock,
    uint32 protection)
```

Creates a new area and returns its **area_id.**

- *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 *)(B_PAGE_SIZE * 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 addressit 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 useful address specification constants, and one that doesn't apply here:

B_EXACT_ADDRESS	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	The area can start at a location equal to or greater than **addr*.
B_ANY_ADDRESS	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).
B_ANY_KERNEL_ADDRESS	The starting address is determined by the system, and the new area will belong to the kernel's team; it won't be deleted when the application quits. In this case, the value that's pointed to by *addr* is ignored (going into the function).
B_CLONE_ADDRESS	This is only meaningful to 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 four locking constants:

B_FULL_LOCK	The area's memory is locked into RAM when the area is created, and won't be swapped out.
B_CONTIGUOUS	Not only is the area's memory locked into RAM, it's also guaranteed to be contiguous. This is particularlyand perhaps exclusivelyuseful to designers of certain types of device drivers.
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	Pages are never locked, they're swapped in and out as needed.
B_LOMEM	This is a special constant that's used for for areas that need to be locked, contiguous, and that fit within the first 16MB of physical memory. The folks that need this constant know who they are.

- *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.

RETURN CODES

- **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 system structures that support this area (unlikely), not enough physical memory to support a locked area, or not enough swap space to allocate virtual memory (in other words, *size* is too big.)
- **B_ERROR**. Some other system error prevented the area from being created.

See also: **clone_area()**

delete_area()

status_t **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. After being deleted, the *area* value is invalid as an area identifier.

Currently, anybody can delete any areathe 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.

RETURN CODES

- **B_OK**. The area was deleted; *area* is now invalid.
- **B_ERROR**. *area* doesn't designate an actual 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 uniquesuccessive 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).

RETURN CODES

- **B_NAME_NOT_FOUND**. The argument doesn't identify an existing area.

See also: **area_for()**

get_area_info() , get_next_area_info() , area_info

status_t **get_area_info(**area_id *area*, area_info **info***)**

status_t **get_next_area_info(**team_id *team*, int32 **cookie*, area_info **info***)**

struct {} **area_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 **get_next_area_info()** version lets you step through the list of a team's areas through iterated calls on the function. The *team* argument identifies the team you want to look at; a *team* value of 0 means the team of the calling thread. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns **B_BAD_VALUE** when there are no more areas to visit:

```
/* Get the area_info for every area in this team. */
area_info info;
int32 cookie = 0;

while (get_next_area_info(0, &cookie, &info) == B_OK)
   ...
```

The **area_info** structure is:

```
typedef struct area_info {
        area_id area;
        char name[B_OS_NAME_LENGTH];
        size_t size;
        uint32 lock;
        uint32 protection;
        team_id team;
        size_t ram_size;
        uint32 copy_count;
        uint32 in_count;
        uint32 out_count;
        void *address;
    } area_info;
```

The 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.
- **size** is the (virtual) 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_CONTIGUOUS, B_LAZY_LOCK**, **B_NO_LOCK**, or **B_LOMEM**.
- **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 team that created or cloned this area.
- **address** is a pointer to the area's starting address. Keep in mind that this address is only meaningful to the team that created (or cloned) the 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 ignoredit 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.

RETURN CODES

- **B_OK**. The area was found; *info* contains valid information.
- **B_BAD_VALUE**. *area* doesn't identify an existing area, *team* doesn't identify an existing team, or there are no more areas to visit.

---

resize_area()

```
status_t resize_area(area_id area, size_t 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 (except, of course, for data that's lost due to shrinkage).

Resizing affects all areas that refer to this areas physical memory. For example, if B is a clone of A, and you resize A, B will be automatically resized (if possible).

RETURN CODES

- **B_OK**. The area was successfully resized.
- **B_BAD_VALUE**. *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 support the new portion of the area. This should only happen if you're increasing the size of the area.
- **B_ERROR**. Some other system error prevented the area from being created.

---

set_area_protection()

status_t **set_area_protection(**area_id *area*, uint32 *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.

RETURN CODES

- **B_OK**. The protection was changed.
- **B_BAD_VALUE**. *area* doesn't identify a valid area.

Area Examples

Example 1: Creating and Writing into an Area

As a simple example of area creation and usage, here we create a ten page area and fill half of it (with nonsense) by bumping a pointer:

```
area_id my_area;
char *area_addr, *ptr;

/* Create an area. */
my_area = create_area("my area", /* name you give to the area */
      (void *)&area_addr,   /* returns the starting addr */
      B_ANY_ADDRESS,        /* area can start anywhere */
      B_PAGE_SIZE*10,       /* size in bytes */
      B_NO_LOCK,            /* Lock in RAM?  No. */
      B_READ_AREA | B_WRITE_AREA); /* permissions */

/* check for errors */
if (my_area < 0) {
      printf("Something bad happenedn");
      return;
}

/* Set ptr to the beginning of the area. */
ptr = area_addr;

/* Fill half the area (with random-ish data). */
for (int i; i < B_PAGE_SIZE*5; i++)
   *ptr++ = system_time()%256;
```

You can also **`memcpy()`** and **`strcpy()`** into the area:

```
/* Copy the first half of the area into the second half. */
memcpy(ptr, area_addr, B_PAGE_SIZE*5);

/* Overwrite the beginning of the area. */
strcpy(area_addr, "Hey, look where I am.");
```

When we're all done, we delete the area:

```
delete_area(my_area);
```

Example 2: Reading a File into an Area

Here's a function that finds a file, opens it (implicit in the BFile constructor), and copies its contents into RAM:

```
#include <File.h>

area_id file_area;

status_t file_reader(const char *pathname)
{
   status_t err;
   char *area_addr;

   BFile file(pathname, B_READ_ONLY);
   if ((err=file.InitCheck()) != B_OK) {
      printf("%s: Can>t find or open.n", pathname);
      return err;
   }

   err = file.GetSize(&file_size);
   if (err != B_OK || file_size == 0) {
      printf("%s:  Disappeared?  Empty?n", pathname);
      return err;
   }

   /* Round the size up to the nearest page. */
   file_size = (((file_size-1)%B_PAGE_SIZE)+1)*B_PAGE_SIZE;

   /* Make sure the size won>t overflow a size_t spec. */
   if (file_size >= ((1<<32)-1) ) {
      printf("%s: What'd you do?  Read Montana?n");
      return B_NO_MEMORY;
   }
   file_area = create_area("File area", (void *)&area_addr,
      B_ANY_ADDRESS, file_size, B_FULL_LOCK,
      B_READ_AREA | B_WRITE_AREA);

   /* Check create_area() errors, as in the last example. */
```

```
   ...

   /* Read the file; delete the area if there>s an error. */
   if ((err=file.Read(area_addr, file_size)) < B_OK) {
      printf("%s: File read error.n");
      delete_area(file_area);
      return err;
   }

   /* The file is automatically closed when the stack-based
    * BFile is destroyed.
    */
   return B_OK;
}
```

Example 3: Accessing a Designated Area

In the previous example, a local variable (**area_addr**) was used to capture the starting address of the newly-created area. If some other function wants to access the area, it must "re-find" the starting address (and the length of the area, for boundary checking). To do this, you call **get_area_info()**.

In the following example, an area is passed in by name; the function, which will write its argument buffer to the area, calls **get_area_info()** to determine the start and extent of the area, and also to make sure that the area is part of this team. If the area was created by some other team, the function could still write to it, but it would have to clone the area first (cloning is demonstrated in the next example).

```
status_t write_to_area(const char *area_name,
               const void *buf,
               size_t len)
{
   area_id area;
   area_info ai;
   thread_id thread;
   thread_info ti;
   status_t err;

   if (!area_name)
      return B_BAD_VALUE;

   area = find_area(area_name);

   /* Did we find it? */
   if (area < B_OK) {
      printf("Couldn>t find area %s.n", area_name);
      return err;
   }

   /* Get the info. */
   err = get_area_info(area, &ai);

   if (err < B_OK) {
      printf("Couldn>t get area info.n");
      return err;
   }

   /* Get the team of the calling thread; to do this, we have
    * to look in the thread_info structure.
    */
   err = get_thread_info(find_thread(NULL), &ti);

   if (err < B_OK) {
      printf("Couldn>t get thread info.n");
      return err;
   }

   /* Compare this team to the area>s team. */
   if (ai.team != ti.team)
      printf("Foreign area.n");
      return B_NOT_ALLOWED;
   }
   /* Make sure we>re not going to overflow the area,
    * and make sure this area can be written to.
    */
   if (len > ai.size) {
      printf("Buffer bigger than area.n");
      return B_BAD_VALUE;
   }
   if (!(ai.protection & B_WRITE_AREA)) {
      printf("Can>t write to this area.n");
      return B_NOT_ALLOWED;
   }

   /* Now we can write. */
   memcpy(ai.address, buf, len);
   return B_OK;
}
```

It's important that you only write to areas that were created or cloned within the calling team. The starting address of a "foreign" area is usually meaningless within your own address space.

You don't *have* to check the area's protectection before writing to it (or reading from it). The memory-accessing fucntions (**memcpy()**, in this example) will segfault if an invalid read or write is requested.

Example 4: Cloning and Sharing an Area

In the following example, a server and a client are set up to share a common area. Here's the server:

```
/* Server side */
class AServer
{
   status_t make_shared_area(size_t size);
   area_id the_area;
   char *area_addr;
};

status_t AServer::make_shared_area(size_t size)
{
   /* The size must be rounded to a page. */
   size = ((size % B_PAGE_SIZE)+1) * B_PAGE_SIZE;
   the_area = create_area("server area", (void *)&area_addr
            B_ANY_ADDRESS, size, B_NO_LOCK,
             B_READ_AREA|B_WRITE_AREA);

   if (the_area < B_OK) {
      printf("Couldn>t create server arean");
      return the_area;

   return B_OK;
}
```

And here's the client:

```
/* Client side */
class AClient
{
   status_t make_shared_clone();
   area_id the_area;
   char *area_addr;
};

status_t AClient::make_shared_clone()
{
   area_id src_area;

   src_area = find_area("server area");
   if (src_area < B_ERROR) {
      printf("Couldn>t find server area.n");
      return src_area;
   }
   the_area = clone_area("client area",
               (void *)&area_addr,
                  B_ANY_ADDRESS,
               B_READ_AREA | B_WRITE_AREA,
               src_area);

   if (the_area < B_OK)
      printf("Couldn>t create clone arean");
      return the_area;
   }
   return B_OK;
}
```

Notice that the area creator (the server in the example) doesn't have to designate the created area as sharable. All areas are candidates for cloning.

After it creates the cloned area, the client's **`area_id`** value (**`AClient::the_area`**) will be different from the server's (**`AServer::the_area`**). Even though **`area_id`** numbers are global, the client should only refer to the server's **`area_id`** number in order to clone it. After the clone, the client talks to the area through its own **`area_id`** (the value passed backed by **`clone_area()`**).

Example 5: Cloning Addresses

It's sometimes useful for shared areas (in other words, a "source" and a clone) to begin at the same starting address. For example, if a client's clone area starts at the same address as the server's original area, then the client and server can pass area−accessing pointers back and forth without having to translate the addresses. Here we modify the previous example to do this:

```
status_t AClient::make_shared_clone()
{
   area_id src_area;

   src_area = find_area("server area");

   if (src_area < B_ERROR) {
      printf("Couldn>t find server area.n");
      return B_BAD_VALUE;
   }

   /* This time, we specify the address that we want the
    * clone to start at.  The B_CLONE_ADDRESS constant
    * does this for us.
    */
   area_addr = src_info.address;
   the_area = clone_area("client area",
               (void *)&area_addr,
                  B_CLONE_ADDRESS,
               B_READ_AREA | B_WRITE_AREA,
               src_area);

   if (the_area < B_OK)
      printf("Couldn>t create clone arean");
```

```
            return the_area;
        }
        return B_OK;
    }
```

Of course, demanding that an area begin at a specific address can be too restrictive; if any of the memory within [area_addr, area_addr + src_info.size] is already allocated, the clone will fail.

Image Concepts

An *image* is compiled code. There are three types of image:

- 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 libraries (**libroot.so**, **libbe.so**, and so on) 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. 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 Tracker, or launched it from the command line. It runs in its own teamit doesn't share the address space of the application from which it was launchedand, 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.

To load an app image, you call the **`load_image()`** function:

```
thread_id load_image(int32 argc,
         const char **argv,
         const char **env)
```

The function's first two arguments identify the app image (file) that you want to launchwe'll return to this in a moment. Having located the file, the function creates a new team, spawns a main thread in that team, and returns the **`thread_id`** of the thread to you. The thread isn't running: To make it run you pass the **`thread_id`** to **`resume_thread()`** or **`wait_for_thread()`** (as explained in "Threads and Teams").

The *argc*/*argv* argument pair is copied and forwarded to the new thread's **`main()`** function:

- The first string in the *argv* array must be the name of the image file that you want to launch; **`load_image()`** uses this string to find the file. 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_image()`** returns (rememberthe array is copied before it's passed to the new thread).
- *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 own 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_image()`** returns (that is, if you allocated it yourselfdon't try to free **`environ`**).

The following example demonstrates a typical use of **`load_image()`**. 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() */

char **arg_v; /* choose a name that doesn>t collide with argv */
int32 arg_c; /* same here vis a vis argc */
thread_id exec_thread;
int32 return_value;
```

Install, in the **`arg_v`** array, the "command line" arguments. Let's pretend we're launching a program found in **/boot/home/apps/adder** that 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, 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 *) * (arg_c + 1));

arg_v[0] = strdup("/boot/home/apps/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_image()`**. After the function returns, it's safe to free the allocated **`arg_v`** array:

```
exec_thread = load_image(arg_c, arg_v, environ);

while (−−arg_c >= 0)
   free(arg_v[arg_c]);

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`** linker flag. Instead, use the **`__declspec()`** directive to export each symbol. The macro is described below. If you're compiling for the PPC, you must also export **`#pragma`** symbols; to do this from the BeIDE, go to the **Linker**/**PEF** portion of the **Settings** window and set "Export Symbols" to "Use #pragma".
- The loader looks for libraries by following the **`LIBRARY_PATH`** environment variable. The default library path looks like this:

```
$ echo $LIBRARY_PATH
%A/lib:/boot/home/config/lib:/boot/beos/system/lib
```

where "%A" means the directory that contains the app that the user is lauching.

Exporting and Importing Symbols

If you're developing a shared library you should explicitly export every global symbol in your library by using the **`__declspec()`** macro. To export a symbol, you declare it thus...

```
__declspec(dllexport) type name
```

...where "_declspec(dllexport)" is literal, and *type* and *name* declare the symbol. Some examples:

```
__declspec(dllexport) char *some_name;
__declspec(dllexport) void some_func() {...}
class __declspec(dllexport) MyView {...}
```

To import these symbols, an app that wants to use your library must "reverse" the declaration by replacing **`dllexport`** with **`dllimport`**:

```
__declspec(dllimport) char *some_name;
__declspec(dllimport) void some_func();
class __declspec(dllimport) MyView;
```

The trouble with this system is that it implies two sets of headers, one for exporting (for building your library) and another for importing (that the library client would use). The Be libraries use macros, defined in be/**BeBuild.h**, that throw the import/export switch so the header files can be unified. For example, here's the macro for **libbe**:

```
#if _BUILDING_be
#define _IMPEXP_BE     __declspec(dllexport)
#else
#define _IMPEXP_BE __declspec(dllimport)
#endif
```

When **libbe** is being built, a private compiler directive defines **`_BUILDING_be`** to be non−zero, and **`_IMPEXP_BE`** exports symbols. When a developer includes **BeBuild.h,** the **`_BUILDING_be`** variable is set to zero, so **`_IMPEXP_BE`** is set to import symbols.

You may want to emulate this system by defining macros for your own libraries. This implies that you have to define a compiler switch (analogous to **_BUILDING_be**) yourself. Set the switch to non−zero when you're building your library, and then set it to zero when you publish your headers for use by library clients.

Creating and Using an Add−on Image

An add−on image is indistinguishable from a shared library image. Creating an add−on is exactly like creating a shared library, a topic that we breezed through above, but with a couple of minor tweaks:

- The loader looks for add−ons by following the paths in the **`ADDON_PATH`** environment variable. The default **`ADDON_PATH`** looks like this:

```
$ echo $ADDON_PATH
%A/add-ons:/boot/home/config/add-ons:/boot/beos/system/add-ons
```

- You have to export your add−on symbols, *and* you also must **`extern`** "C" them. This ensures that the symbol names won't be mangled by the compiler.

Exporting Add−on Symbols

To export your add−on's symbols, declare them thus:

```
extern "C" __declspec(dllexport) void some_func();
extern "C" __declspec(dllexport) int32 some_global_data;
```

To extern a C++ class takes more work. You can't extern the class directly; typically what you do is create (and extern) a C function that covers the class constructor:

```
extern "C" __declspec(dllexport) MyClass *instantiate_my_class();
```

`instantiate_my_class()` is implemented to call the MyClass constructor:

```
MyClass *instantiate_my_class()
{
  return new MyClass();
}
```

Loading an Add−on Image

To load an add−on into your application, you call the **load_add_on()** function. The function takes a pathname (absolute or relative to the current working directory) 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 **/boot/home/add−ons/adder**. The code that loads the add−on would look like this:

```
/* For brevity, we won>t check errors.  */
image_id addon_image;

/* Load the add-on. */
addon_image = load_add_on("/boot/home/add-ons/adder");
```

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:

```
status_t get_image_symbol(image_id image,
        char *symbol_name,
        int32 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, and that the add−on has declared the name as **extern**. In general, using an add−on implies just this sort of cooperation.
- The third argument is a constant that gives the symbol's *symbol type*. There are three types, as given below. If the executable format doesn't distinguish between text and data symbols, then you can use any of these three typesthey'll all be the same. If the format *does* distinguish between text and data, then you can either ask for the specific type, or you can ask for **B_SYMBOL_TYPE_ANY**.

B_SYMBOL_TYPE_DATA	Global data (variables)
B_SYMBOL_TYPE_TEXT	Functions
B_SYMBOL_TYPE_ANY	The symbol lives anywhere

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:

```
extern "C" int32 a1 = 0;
extern "C" int32 a2 = 0;
extern "C" int32 adder(void);

int32 adder(void)
{
   return (a1 + a2);
}
```

To examine the variables (**a1** and **a2**), you would call **get_image_symbol()** thus:

```
int32 *var_a1, *var_a2;

get_image_symbol(addon_image, "a1", B_SYMBOL_TYPE_DATA, &var_a1);
get_image_symbol(addon_image, "a2", B_SYMBOL_TYPE_DATA, &var_a2);
```

Here we get the symbol for the **adder()** function:

```
int32 (*func_add)();
get_image_symbol(addon_image, "adder", B_SYMBOL_TYPE_TEXT, &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;
int32 return_value = (*func_add)();
```

Images

Declared in: be/kernel/image.h

Library: libroot.so

This isn't about graphics. An *image* is compiled code, of which there are three types: app images, library images, and add−on images. An app image is executable code that can be launched. A library image is a collection of shared code that you link against when you're compiling your application. An add−on image is code that an app can load and run while the app itself is running. Note that an add−on can also be an app; in other words, you can create an image that can be launched by itself, or that can be loaded into another application.

For more information on creating and using images, see "Image Concepts".

Image Functions

get_image_info() , get_next_image_info() , image_info

```
status_t get_image_info(image_id image, image_info *info)

status_t get_next_image_info(team_id team,
    int32 *cookie,
    image_info *info)

struct {} image_info
```

These functions copy, into the *info* argument, the **image_info** structure for a particular image. The **get_image_info()** function gets the information for the image identified by *image*.

The **get_next_image_info()** function lets you step through the list of a team's images through iterated calls. The *team* argument identifies the team you want to look at; a *team* value of 0 means the team of the calling thread. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns **B_BAD_VALUE** when there are no more images to visit:

```
/* Get the image_info for every image in this team. */
image_info info;
int32 cookie = 0;

while (get_next_image_info(0, &cookie, &info) == B_OK)
   ...
```

The **image_info** structure is:

```
typedef struct {
      image_id id;
      image_type type;
      int32 sequence;
      int32 init_order;
      B_PFV init_routine;
      B_PFV term_routine;
      dev_t device;
      ino_t node;
      char name[MAXPATHLEN];
      void *text;
      void *data;
      int32 text_size;
      int32 data_size;
   } image_info
```

The fields are:

- **id**. The image's **image_id** number.
- **type**. A constant (listed below) that tells whether this is an app, library, or add−on image.
- **sequence** and **init_order**. These are zero−based ordinal numbers that give the order in which the image was loaded and initialized, compared to all the other images in this team.
- **init_routine** and **term_routine**. These are pointers to the functions that are used to intialize and terminate the image (more specifically, the image's main thread). The **B_PFV** type is a cover for a pointer to a **(void***) function.
- **device**. The device that the image file lives on.
- **node**. The node number of the image file.
- **name**. The full pathname of the file whence sprang the image.

- **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.

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

B_APP_IMAGE
B_LIBRARY_IMAGE
B_ADD_ON_IMAGE

RETURN CODES

B_OK. The image was found; *info* contains valid information.

- **B_BAD_VALUE**. *image* doesn>t identify an existing image, *team* doesn>t identify an existing team, or there are no more images to visit.

---

get_image_symbol() , get_nth_image_symbol()

```
status_t get_image_symbol(image_id image,
    char *symbol_name,
    int32 symbol_type,
    void **location)

status_t get_nth_image_symbol(image_id image,
    int32 n,
    char *name,
        int32 *name_length,
    int32 *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."

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 inthe 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.

  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, **B_SYMBOL_TYPE_TEXT** if the symbol is a function, or **B_SYMBOL_TYPE_ANY** if the executable format doesn't distinguish between the two. 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_next_image_info()** function. Such numbers are also returned directly when you load an add−on image through the **load_add_on()** function.

RETURN CODES

B_OK. The symbol was found.

- **B_BAD_IMAGE_ID**. *image* doesn>t identify an existing image.
- **B_BAD_INDEX**. *n* is out−of−bounds.

---

load_add_on() , unload_add_on()

```
image_id load_add_on(const char *pathname)

status_t unload_add_on(image_id image)
```

load_add_on() loads an add−on image, identified by *pathname*, into your application's address space.

- *pathname* can be absolute or relative; if it's relative, it's reckoned off the base path specified by the **ADDON_PATH** environment variable.
- The function returns an **image_id** (a positive integer) that represents the loaded image. Image ID numbers are unique across the system.

An example that demonstrates the use of **load_add_on()** is given in "Loading an Add−on Image."

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.

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_OK**.

RETURN CODES

Positive **image_id *value*** (load) *or* **B_OK** (unload). Success.

- **B_ERROR**. The image couldn't be loaded (for whatever reason), or *image* isn't a valid image ID.

---

load_image()

thread_id **load_image(**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 pathname given in *argv*[0].

The 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. 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_image(..., 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_image()** returns.

The thread that's returned by **load_image()** 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_image()** is given in "Loading an App Image."

RETURN CODES

- Positive integers. Success.
- **B_ERROR**. Failure, for whatever reason.

Port Concepts

A port is a system−wide message repository into which any thread can copy a buffer of data, and from which any 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.

Creating and Destroying a Port

The **create_port()** function creates a new port and assigns it a unique, system−wide **port_id** number. Although ports are accessible to all threads, the **port_id** numbers aren't disseminated by the operating systemthere's no "find port" function. 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.

A port is owned by the team in which it was created. When a team dies (when all its threads are killed), 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 call **delete_port()**. You can delete any port, not just those that are owned by the team of the calling thread. When you delete a port, all of its unread messages are thrown away. If you want to read this messages, but you don't want any new messages to arrive in the meantime, you should call **close_port()** before deleting the port. Note that you can't reopen a closed port; after you get done reading the port's messages, you're expected to delete the port.

The Message Queue: Reading and Writing Port Messages

The length of a port's message queuethe number of messages that it can hold at a timeis set when the port is created.

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.

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" poolthere 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 messagethe data that's sent through a portconsists 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 lostthe 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 = NULL;
ssize_t size;
int32 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_OK)
   /* Handle the error */

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

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

Obviously, there's a race condition (in the example) between **port_buffer_size()** and the subsequent **read_port()** callsome 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 at a time.

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.

Ports

Declared in: be/kernel/OS.h

Library: libroot.so

A port is a system−wide message repository into which any thread can copy a buffer of data, and from which any 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.

Ports are largely subsumed by the Application Kit's BMessage class (and relatives). The two features of ports that you can't get at the BMessage level are:

- Ports let you set the length of the message queue.
- Ports can be used in C code (as opposed to C++).

For most applications, these are inessential additions.

For more information on ports, see "Port Concepts".

Port Functions

create_port()

```
port_id create_port(int32 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 filledwhen it's holding *queue_length* messagessubsequent invocations of **write_port()** (on that port) block until room is made in the queue (through calls to **read_port()**) for the additional messages. Once the queue length is set by **create_port()**, it can't be changed.

This function 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.

RETURN CODES

- **B_BAD_VALUE**. *queue_length* is too big or less than zero.
- **B_NO_MORE_PORTS**. The system couldn't allocate another port.

close_port()

```
status_t close_port(port_id port)
```

Closes the *port* so no more messages can be written to it. After you close a port, you can call **read_port()** on it, but a **write_port()** call will return **B_BAD_PORT_ID**. You can't reopen a closed port; you call this function so you can read through a port's unread messages prior to deleting the port, while ensuring that no new messages will show up. After you've read the messages, you should call **delete_port()** on *port*.

RETURN CODES

- **B_OK**. *port* is now closed.
- **B_BAD_PORT_ID**. *port* doesn't identify an open port.

delete_port()

```
status_t delete_port(port_id port)
```

Deletes the given port. The port's message queue doesn't have to be emptyyou 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.

RETURN CODES

- **B_OK.** The port was deleted.
- **B_BAD_PORT_ID**. *port* isn't a valid port.

---

find_port()

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

Returns the **port_id** of the named port. port_name should be no longer than 32 characters (**B_OS_NAME_LENGTH**).

RETURN CODES

- **B_NAME_NOT_FOUND**. *port_name* doesn't name an existing port.

---

get_port_info() , get_next_port_info()

status_t **get_port_info(port_id** *port*, **port_info** **info***)*

status_t **get_next_port_info(team_id** *team*,
uint32 **cookie*,
port_info **info***)*

Copies information about a particular port into the **port_info** structure designated by *info*. The first version of the function designates the port directly, by **port_id**.

The **get_next_port_info()** version lets you step through the list of a team's ports through iterated calls on the function. The *team* argument identifies the team you want to look at; a *team* value of 0 means the team of the calling thread. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns **B_BAD_VALUE** when there are no more ports to visit:

```
/* Get the port_info for every port in this team. */
port_info info;
int32 cookie = 0;

while (get_next_port_info(0, &cookie, &info) == B_OK)
   ...
```

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.

RETURN CODES

- **B_OK**. The port was found; *info* contains valid information.
- **B_BAD_VALUE**. *port* doesn't identify an existing port, *team* doesn't identify an existing team, or there are no more ports to visit.

---

port_buffer_size() , port_buffer_size_etc()

ssize_t **port_buffer_size(**port_id *port***)**

ssize_t **port_buffer_size_etc(**port_id *port*,
uint32 *flags*,
bigtime_t *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.

RETURN CODES

- **B_BAD_PORT_ID**. *port* doesn't identify an existing port, or the port was deleted while the function was blocked.
- **B_TIMED_OUT**. The *timeout* limit expired.

- **B_WOULD_BLOCK**. You asked for a *timeout* of 0, but there are no messages in the queue.

See also: read_port()

port_count()

int32 **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.

RETURN CODES

- **B_BAD_PORT_ID**. *port* doesn't identify an existing port.

See also: get_port_info()

read_port() , read_port_etc()

ssize_t **read_port(**port_id *port*,
int32 **msg_code*,
void **msg_buffer*,
size_t *buffer_size***)**

ssize_t **read_port_etc(**port_id *port*,
int32 **msg_code*,
void **msg_buffer*,
size_t *buffer_size*,
uint32 *flags*,
bigtime_t *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.

RETURN CODES

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

- **B_BAD_PORT_ID**. *port* doesn't identify an existing port, or the port was deleted while the function was blocked.
- **B_TIMED_OUT**. The *timeout* limit expired.
- **B_WOULD_BLOCK**. You asked for a *timeout* of 0, but there are no messages in the queue.

See also: write_port(), port_buffer_size()

set_port_owner()

status_t **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 freed. Ownership, otherwise, has no significanceit carries no special privileges or obligations.

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

RETURN CODES

- **B_OK**. Ownership was successfully transferred.
- **B_BAD_PORT_ID**. *port* doesn't identify a valid port.
- **B_BAD_TEAM_ID**. *team* doesn't identify a valid team.

See also: **get_port_info()**

---

write_port() , write_port_etc()

```
status_t write_port(port_id port,
    int32 msg_code,
    void *msg_buffer,
    size_t buffer_size)

status_t write_port_etc(port_id port,
    int32 msg_code,
    void *msg_buffer,
    size_t buffer_size,
    uint32 flags,
    bigtime_t 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 or closed.

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.

RETURN CODES

- **B_OK**. The port was successully written to.
- **B_BAD_PORT_ID**. *port* doesn't identify an open port, or the port was deleted while the function was blocked.
- **B_TIMED_OUT**. The *timeout* limit expired.
- **B_WOULD_BLOCK**. You asked for a *timeout* of 0, but there are no free slots in the message queue.

See also: **read_port()**

Port Structures and Constants

port_info

```
struct {
    port_id port;
    team_id team;
    char name[B_OS_NAME_LENGTH];
    int32 capacity;
    int32 queue_count;
    int32 total_count;
} port_info
```

The port_info structure provides information about a port. You retrieve one of these structures through **get_port_info()** or **get_next_port_info()**.

- port. The **port_id** number of the port.
- team. The **team_id** of the port's owner.
- 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.

Semaphores

Declared in: be/kernel/OS.h

Library: libroot.so

A semaphore is a token that's used to synchronize multiple threads. The semaphore concept is simple: To enter into a semaphore−protected "critical section", a thread must first "acquire" the semaphore, through the **acquire_sem()** function. When it passes out of the critical section, the thread "releases" the semaphore through **release_sem()**.

The advantage of the semaphore system is that if a thread can't acquire a semaphore (because the semaphore is yet to be released by the previous acquirer), the thread blocks in the **acquire_sem()** call. While it's blocked, the thread doesn't waste any cycles.

For the full story about semaphores, see "Semaphore Concepts". For some code examples, see "Semaphore Examples".

Semaphore Functions

acquire_sem() , acquire_sem_etc()

```
status_t acquire_sem(sem_id sem)

status_t acquire_sem_etc(sem_id sem,
    uint32 count,
    uint32 flags,
    bigtime_t 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 add **B_ABSOLUTE_TIMEOUT** or **B_RELATIVE_TIMEOUT** to the *flags* argument. *timeout* to the amount of time, in microseconds, that you're willing to wait, measured relative to now (relative timeout), or in comparison to the value returned by system_time() (absolute timeout). The function returns **B_TIMED_OUT** if the semaphore isn't acquired within the specified time. If you specify a relative *timeout* of 0 and the semaphore isn't immediately available, the function immediately returns **B_WOULD_BLOCK**.

The Kernel Kit defines two other semaphore−acquisition flag constants (**B_CAN_INTERRUPT** and **B_CHECK_PERMISSION**). These additional flags are used by device driversadding 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_etc()**) 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).

RETURN CODES

- **B_NO_ERROR**. The semaphore was successfully acquired.
- **B_BAD_SEM_ID**. The *sem* argument doesn't identify a valid semaphore. 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()**.
- **B_INTERRUPTED**. The acquisition was interrupted by a signal. In this case, the semaphore has *not* been acquired.

The other return values apply to **acquire_sem_etc()** only:

- **B_BAD_VALUE**. Illegal *count* value (less than 1).

- **B_WOULD_BLOCK**. You specified a relative *timeout* of 0 and the semaphore isn't available.
- **B_TIMED_OUT**. The timeout expired (for all values of *timeout* other than 0).

---

create_sem()

sem_id **create_sem(**uint32 *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.
- *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 uniqueany 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_OK)
   /* 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.

RETURN CODES

- **B_BAD_VALUE**. Invalid *thread_count* value (less than 0).
- **B_NO_MEMORY**. Not enough memory to allocate the semaphore's name.
- **B_NO_MORE_SEMS**. All valid **sem_id** numbers are being used.

---

delete_sem()

status_t **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 unblocked.

This function may only be called from a thread that belongs to the semaphore's owner.

RETURN CODES

- **B_NO_ERROR**. The semaphore was successfully deleted.
- **B_BAD_SEM_ID**. *sem* is invalid, or the calling thread doesn't belong to the team that owns the semaphore.

---

get_sem_count()

status_t **get_sem_count(**sem_id *sem*, int32 **thread_count***)*

For amusement purposes only; never predicate your code on this function.

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.

RETURN CODES

- **B_NO_ERROR**. Success.
- **B_BAD_SEM_ID**. *sem* is invalid(*thread_count* isn't changed).

---

get_sem_info() , get_next_sem_info()

status_t **get_sem_info(**sem_id *sem*, sem_info **info***)**

status_t **get_next_sem_info(**team_id *team*,
uint32 *cookie,
sem_info **info***)**

Copies information about a particular semaphore into the **sem_info** structure designated by *info*. The first version of the function designates the sempahore directly, by **sem_id**.

The **get_next_sem_info()** version lets you step through the list of a team's semaphores through iterated calls on the function. The *team* argument identifies the team you want to look at; a *team* value of 0 means the team of the calling thread. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns **B_BAD_VALUE** when there are no more sempahores to visit:

```
/* Get the sem_info for every sempahore in this team. */
sem_info info;
int32 cookie = 0;

while (get_next_sem_info(0, &cookie, &info) == B_OK)
   ...
```

RETURN CODES

- **B_NO_ERROR**. Success.
- **B_BAD_SEM_ID**. Invalid *sem* value.
- **B_BAD_TEAM_ID**. Invalid *team* value.

---

release_sem() , release_sem_etc()

status_t release_sem(sem_id *sem***)**

status_t release_sem_etc(sem_id *sem*, **int32** *count*, **uint32** *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()**, 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.

RETURN CODES

- **B_NO_ERROR**. The semaphore was successfully released.
- **B_BAD_SEM_ID**. Invalid *sem* value.
- **B_BAD_VALUE**. Invalid *count* value (less than zero; **release_sem_etc()** only).

See also: **acquire_sem()**

---

set_sem_owner()

```
status_t 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.
- Threads can only by deleted by threads that belongs to a semaphore's owner.

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

RETURN CODES

- **B_NO_ERROR**. Ownership was successfully transferred.
- **B_BAD_SEM_ID**. Invalid *sem* value.
- **B_BAD_TEAM_ID**. Invalid *team* value.

Semaphore Structures and Types

sem_id

```
typedef int32 sem_id;
```

sem_id numbers identify semaphores. The id is assigned when the semaphore is created (**create_sem()**). The values are unique across the system.

sem_info

```
typedef struct sem_info {
        sem_id sem;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        int32 count;
        thread_id latest_holder;
    }
```

The sem_info structure supplies information about a semaphore. You retrieve the structure through the **get_sem_info()** function. 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 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.

The lastest_holder field is *highly* undependable; in some cases, the kernel doesn't even record the semaphore acquirer. Although you can use this field as a hint while debugging, you shouldn't take it too seriously. Love, Mom.

Semaphore Constants

Semaphore Control Flags

- **B_CAN_INTERRUPT** Tells the kernel that the semaphore can be interrupted by a signal.
- **B_DO_NOT_RESCHEDULE** Tells the scheduler not to run after a semaphore is released. In other words, the thread that just released the semaphore gets to keep running.
- **B_CHECK_PERMISSION** Makes sure that the semaphore acquirer/releaser is running at the proper level. This is always added into user−level acquisition and release.
- **B_RELATIVE_TIMEOUT** Used to set a timeout that's relative to now.
- **B_ABSOLUTE_TIMEOUT** Used to set a timeout that's measured against the system clock.
- **B_TIMEOUT** Obsolete; use `B_RELATIVE_TIMEOUT` .

Semaphore Concepts

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`** identifiera system−wide number that's assigned when the semaphore is createdto the **`acquire_sem()`** function. The function blocks until the semaphore is actually acquired.

An alternate function, **`acquire_sem_etc()`** lets you specify the amount of time you're willing to wait for the semaphore to be acquired, and let you acquire the semaphore more than once in a single go. 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. 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:

```
if (acquire_sem(my_semaphore) == B_NO_ERROR) {
   /* Protected code goes here. */
   release_sem(my_semaphore);
}
```

Keep in mind that...

- The calls to the acquire and release functions needn't be locally balanced (although this is by far the most common use). 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.
- Checking the value returned by **`acquire_sem()`** is *extremely* important. If an acquire−blocked thread is unblocked by a signal (a return o f **`B_INTERRUPTED`**), the thread shouldn't procede to the critical section.

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 where it sits blocked in the **`acquire_sem()`** call. Each call to **`release_sem()`** umblocks the thread at the head of that semaphore's queue, thus allowing the thread to return from its call to **`acquire_sem()`**.

Semaphores don't discriminate between acquisitive threadsthey don't prioritize or otherwise reorder the threads in their queuesthe 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. Ostensibly, 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. (As we'll see later, this isn't the entire story, but it's good enough for now.) For example, a semaphore that's used as a mutually exclusive lock takes an initial thread count of 1in other words, only one thread can acquire the semaphore at a time.

An initial thread count of 1 is by far the most common use; a thread count of 0 is also useful. Other counts are much less common.

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 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 semaphoreit'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 commonit'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 amusementwhile you're debugging, for example.

You should never predicate your code on the basis of a semaphore's thread count.

Deleting a Semaphore

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 semaphoreyou 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: When you delete a semaphore (or when it dies naturally), all its queued threads are immediately allowed to continuethey 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).

Inter−application Semaphores

The **sem_id** number that identifies a semaphore is a system−wide tokenthe **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 thembut 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 accept messages from other applications and then spawn threads that acquire and release the appropriate semaphores.

Semaphore Examples

The following sections provides examples of typical semaphore use. For the full story on semaphores, see Semaphores.

Semaphore Example 1: Locking

The most typical use of a semaphore is to protect a chunk of code that can only be executed by one thread at a time. The semaphore acts as a lock; **`acquire_sem()`** locks the code, **`release_sem()`** releases it. Semaphores that are used as locks are (almost always) created with a thread count of 1.

As a simple example, let's say you keep track of a maximum value like this:

```
/* max_val is a global. */
uint32 max_val = 0;
...

/* bump_max() resets the max value, if necessary. */
void bump_max(uint32 new_value)
{
   if (new_value > max_value)
      max_value = new_value;
}
```

bump_max() isn't thread safe; there's a race condition between the comparison and the assignment. So we protect it with a semaphore:

```
sem_id max_sem;
uint32 max_val = 0;

...
/* Initialize the semaphore during a setup routine. */
status_t init()
{
   if ((max_sem = create_sem(1, "max_sem")) < B_NO_ERROR)
      return B_ERROR;
   ...
}
void bump_max(uint32 new_value)
{
   if (acquire_sem(max_sem) != B_NO_ERROR)
      return;
   if (new_value > max_value)
      max_value = new_value;
   release_sem();
}
```

Semaphore Example 2: Benaphores

A "benaphore" is a combination of an atomic variable and a semaphore that can improve locking efficiency. If you're using a semaphore as shown in the previous example, you should consider using a benaphore instead (if you can).

Here's the example re−written to use a benaphore:

```
sem_id max_sem;
uint32 max_val = 0;
int32 ben_val = 0;

status_t init()
{
   /* This time we initialized the semaphore to 0. */
   if ((max_sem = create_sem(0, "max_sem")) < B_NO_ERROR)
      return B_ERROR;
   ...
}
void bump_max(uint32 new_value)
{
   int32 previous = atomic_add(&ben_val, 1);
   if (previous >= 1)
      if (acquire_sem(max_sem) != B_NO_ERROR)
         goto get_out;

   if (new_value > max_value)
      max_value = new_value;

get_out:
   previous = atomic_add(&ben_val, −1);
   if (previous > 1)
      release_sem(max_sem);
}
```

The point, here, is that **`acquire_sem()`** is called only if it's known (by checking the previous value of **`ben_val`**) that some other thread is in the middle of the critical section. On the releasing end, the **`release_sem()`** is called only if some other thread has since entered the function (and is now blocked in the **`acquire_sem()`** call). An important point, here, is that the semaphore is initialized to 0.

Semaphore Example 3: Imposing 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, we have a global buffer that's accessed through separate reading and writing functions. Furthermore, we want writes

and reads to alternate, with a write going first.

We can lock the entire buffer with a single semaphore, but to enforce alternation we need two semaphores:

```
sem_id write_sem, read_sem;
char buffer[1024];

/* Initialize the semaphores */
status_t init()
{
   if ((write_sem = create_sem(1, "write")) < B_NO_ERROR) {
      return;
   if ((read_sem = create_sem(0, "read")) < B_NO_ERROR) {
      delete_sem(write_sem);
      return;
   }
}

status_t write_buffer(const char *src)
{
   if (acquire_sem(write_sem) != B_NO_ERROR)
      return B_ERROR;

   strncpy(buffer, src, 1024);

   release_sem(read_sem);
}

status_t read_buffer(char *dest, size_t len)
{
   if (acquire_sem(read_sem) != B_NO_ERROR)
      return B_ERROR;

   strncpy(dest, buffer, len);

   release_sem(write_sem);
}
```

The initial thread counts ensure that the buffer will be written to before it's read: If a reader arrives before a writer, the reader will block until the writer releases the **`read_sem`** semaphore.

System Information

Declared in: be/kernel/OS.h

Library: libroot.so

The following functions, types, and structures convey information about the system, such as the number of CPUs, when the kernel was built, and whether your computer is on fire.

System Info Functions

get_system_info()

```
status_t get_system_info(system_info *info)
```

The `get_system_info()` function tells you more than you want to know about the physical capacities of your computer and other statistics of your operating system. The function copies this information into the `system_info` argument *info*. You must allocate *info* before passing it in.

RETURN CODES

`get_system_info()` always returns `B_OK`.

is_computer_on()

```
int32 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.

is_computer_on_fire()

```
double is_computer_on_fire(void)
```

Returns the temperature of the motherboard if the computer is currently on fire. If the computer isn't on fire, the function returns some other value.

System Info Structures and Constants

cpu_info

```
typedef struct {
        bigtime_t active_time;
} cpu_info
```

The `cpu_info` structure provides information about your computer's CPU(s). You retrieve the structure by looking in the cpu_infos field of the `system_info` structure. The cpu_infos field is an array that contains one `cpu_info` structure for each CPU on your motherboard.

`active_time`	The number of microseconds the CPU has spent doing useful work (i.e. not busy waiting) since the machine was booted.

cpu_type

```
typedef enum cpu_types {
    B_CPU_PPC_601,
    B_CPU_PPC_603,
    B_CPU_PPC_603e,
```

```
    B_CPU_PPC_604,
    B_CPU_PPC_604e,
    B_CPU_PPC_750,
    B_CPU_PPC_686,
    B_CPU_AMD_29K,
    B_CPU_X86, Obsolete
        B_CPU_MC6502,
    B_CPU_Z80,
    B_CPU_ALPHA,
    B_CPU_MIPS,
    B_CPU_HPPA,
    B_CPU_M68K,
    B_CPU_ARM,
    B_CPU_SH,
    B_CPU_SPARC,

    B_CPU_INTEL_X86 = 0x1000,
        B_CPU_INTEL_PENTIUM,
    B_CPU_INTEL_PENTIUM75,
    B_CPU_INTEL_PENTIUM_486_OVERDRIVE,
    B_CPU_INTEL_PENTIUM_MMX,
    B_CPU_INTEL_PENTIUM_MMX_MODEL_4,
    B_CPU_INTEL_PENTIUM_MMX_MODEL_8,
    B_CPU_INTEL_PENTIUM75_486_OVERDRIVE,
    B_CPU_INTEL_PENTIUM_PRO,
    B_CPU_INTEL_PENTIUM_II,
    B_CPU_INTEL_PENTIUM_II_MODEL_3,
    B_CPU_INTEL_PENTIUM_II_MODEL_5,
    B_CPU_INTEL_CELERON,
    B_CPU_INTEL_PENTIUM_III,

    B_CPU_AMD_X86 = 0x1100,
    B_CPU_AMD_K5_MODEL0,
    B_CPU_AMD_K5_MODEL1,
    B_CPU_AMD_K5_MODEL2,
    B_CPU_AMD_K5_MODEL3,
    B_CPU_AMD_K6_MODEL6,
    B_CPU_AMD_K6_MODEL7,
    B_CPU_AMD_K6_MODEL8,
    B_CPU_AMD_K6_2,
    B_CPU_AMD_K6_MODEL9,
    B_CPU_AMD_K6_III,
    B_CPU_AMD_ATHLON_MODEL1,

    B_CPU_CYRIX_X86 = 0x1200,
    B_CPU_CYRIX_GXm,
    B_CPU_CYRIX_6x86MX,

    B_CPU_IDT_X86 = 0x1300,
    B_CPU_IDT_WINCHIP_C6,
    B_CPU_IDT_WINCHIP_2,

    B_CPU_RISE_X86 = 0x1400,
    B_CPU_RISE_mP6
} cpu_type;
```

The **B_X86_CPU** constant is obsolete. All x86 CPUs are represented by specific vendor/model constants.

These constants represents different CPU vendors and models. To retrieve the constant that represents the CPU(s) that your computer uses, look in the cpu_type field of the **system_info** structure.

Note that the x86 types are grouped by vendor, where the vendor is identified by the high word, and the model by the low word. Each vendor has its own **cpu_type** constant, as indicated by the explicit values in the list above. If you're only interested in the vendor, mask the **cpu_type** value with **B_CPU_X86_VENDOR_MASK**, thus:

```
system_info sysinfo;
get_system_info( &sysinfo );

switch ( sysinfo.cpu_type & B_CPU_X86_VENDOR_MASK ) {
   case B_CPU_INTEL_X86: ...
   case B_CPU_AMD_X86: ...
   case B_CPU_CYRIX_x86: ...
   case B_CPU_IDT_X86: ...
   case B_CPU_RISE_X86: ...
   default:
   /* Not x86. */
}
```

B_CPU_X86_VENDOR_MASK

```
B_CPU_X86_VENDOR_MASK
```

You mask your machine's **cpu_type** value with this constant to retrieve the CPU's vendor id (which is also a **cpu_type** value). If the CPU isn't an breed of x86, applying the mask will yield 0. See the **cpu_type** description for an example.

---

machine_id

```
typedef int32 machine_id[2]; Currently unused
```

machine_id is a 64−bit number (encoded as two 32−bit numbers) that uniquely identifies this specific computer. The id number is returned as the id field (*currently unused*) of the **system_info** structure.

B_MAX_CPU_COUNT

```
B_MAX_CPU_COUNT
```

This constant is set to the maximum number of CPUs that the BeOS can take advantage of. The number is machine−dependent.

---

platform_type

```
typedef enum platform_types {
    B_BEBOX_PLATFORM = 0,
    B_MAC_PLATFORM,
    B_AT_CLONE_PLATFORM,
    B_ENIAC_PLATFORM,
    B_APPLE_II_PLATFORM,
    B_CRAY_PLATFORM,
    B_LISA_PLATFORM,
    B_TI_994A_PLATFORM,
    B_TIMEX_SINCLAIR_PLATFORM,
    B_ORAC_1_PLATFORM,
    B_HAL_PLATFORM,
    B_BESM_6_PLATFORM,
    B_MK_61_PLATFORM,
    B_NINTENDO_64_PLATFORM
    } platform_type;
```

These constants represent the various computer platforms that the BeOS does, has, should, might, or might not run on. To retrieve the constant for the machine that you're running on, look in the platform_type field of the **system_info** structure.

---

system_info

```
typedef struct {
        machine_id id; Currently unused
        bigtime_t boot_time;
    int32 cpu_count;
        cpu_type cpu_type;
    int32 cpu_revision;
        cpu_info cpu_infos[B_MAX_CPU_NUM];
    int64 cpu_clock_speed;
    int64 bus_clock_speed;
        platform_type platform_type;
    int32 max_pages;
    int32 used_pages;
    int32 page_faults;
    int32 max_sems;
    int32 used_sems;
    int32 max_ports;
    int32 used_ports;
```

```
int32 max_threads;
int32 used_threads;
int32 max_teams;
int32 used_teams;
char kernel_name[B_FILE_NAME_LENGTH];
char kernel_build_date[B_OS_NAME_LENGTH];
char kernel_build_time[B_OS_NAME_LENGTH];
int64 kernel_version;
} system_info;
```

The **system_info** structure describes your computer's hardware and operating system, and provides information about the availability of kernel resources (such as threads and ports). You retrieve a system_info structure through the **get_system_info()** function.

id	A 64−bit number (encoded as two **int32**s) that uniquely identifies this machine. *Currently unsed.*
boot_time	The time at which the computer was last booted, measured in microseconds since January 1st, 1970. You can also get this information through **system_time()**.
cpu_count	The number of CPUs on your computer's motherboard.
cpu_type	A constant that represents the CPU(s) make and model.
cpu_revision	The revision number of the CPU(s).
cpu_infos	An array of **cpu_info** structures, one for each CPU.
cpu_clock_speed	The speed (in Hz) at which the CPU(s) operate.
bus_clock_speed	The speed (in Hz) at which the bus operates.
platform_type	The platform type constant that reprsents your computer.
max_resource used_resource	These 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 has read a page of memory into RAM due to a page fault.
kernel_name	The filename of the kernel (the leaf name, not the path).
kernel_build_date	Fixed−width string that

	gives the date that the kernel was built; for example: `Jun 5 1999`
kernel_build_time Fixed−width string that gives the time of day that the kernel was built; for example: 10:02:16	
kernel_version A Be−assigned number that identifies the kernel version.	

System and Time Information

Declared in: be/kernel/OS.h

Library: libroot.so

The following functions, types, and structures are used to convey basic information about the system, such as the number of CPUs, when the kernel was built, what time it is now and whether your computer is on fire.

System Info Functions and Structures

get_system_info() , system_info , cpu_info , cpu_type , platform_type

status_t **get_system_info(**system_info **info**)

struct {} **system_info**

struct {} **cpu_info**

enum **cpu_type**

enum **platform_type**

The **get_system_info()** function tells you more than you want to know about the physical capacities and other statistics of your operating system. The function takes a pointer to an allocated **system_info** structure and fills it in.

```
typedef struct {
   machine_id  id;
   bigtime_t  boot_time;
   int32  cpu_count;
   cpu_type  cpu_type;
   int32  cpu_revision;
   cpu_info  cpu_infos[B_MAX_CPU_NUM];
   int64  cpu_clock_speed;
   int64  bus_clock_speed;
   platform_type  platform_type;
   int32  max_pages;
   int32  used_pages;
   int32  page_faults;
   int32  max_sems;
   int32  used_sems;
   int32  max_ports;
   int32  used_ports;
   int32  max_threads;
   int32  used_threads;
   int32  max_teams;
   int32  used_teams;
   char  kernel_name[B_FILE_NAME_LENGTH];
   char  kernel_build_date[B_OS_NAME_LENGTH];
   char  kernel_build_time[B_OS_NAME_LENGTH];
   int64  kernel_version;
} 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 **int32**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.
- **platform_type**. One of the platform type constants.
- **max_*resources*** and **used_*resources***. 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.
- **kernel_name**. The (leaf) name of the kernel.
- **kernel_build_date** and **kernel_build_time**. Human−readable strings that tell you when the kernel was built.
- **kernel_version.** A number that identifies the kernel version.

The **cpu_info** structure is:

```
typedef struct {
   bigtime_t active_time;
} cpu_info;
```

- **active_time** is the number of microseconds spent doing useful work since the machine was booted.

Relatedly, **B_MAX_CPU_COUNT** is currently 8.

The **machine_id** type is:

```
typedef int32 machine_id[2];
```

The **cpu_type** constants are:

```
typedef enum {
    B_CPU_PPC_601 = 1,
    B_CPU_PPC_603 = 2,
    B_CPU_PPC_603e = 3,
    B_CPU_PPC_604 = 4,
    B_CPU_PPC_604e = 5,
    B_CPU_PPC_686 = 13,
    B_CPU_AMD_29K,
    B_CPU_X86,
    B_CPU_MC6502,
    B_CPU_Z80,
    B_CPU_ALPHA,
    B_CPU_MIPS,
    B_CPU_HPPA,
    B_CPU_M68K,
    B_CPU_ARM,
    B_CPU_SH,
    B_CPU_SPARC
} cpu_type;
```

The **platform_type** constants are:

```
typedef enum {
    B_BEBOX_PLATFORM = 0,
    B_MAC_PLATFORM,
    B_AT_CLONE_PLATFORM,
    B_ENIAC_PLATFORM,
    B_APPLE_II_PLATFORM,
    B_CRAY_PLATFORM,
    B_LISA_PLATFORM,
    B_TI_994A_PLATFORM,
    B_TIMEX_SINCLAIR_PLATFORM,
    B_ORAC_1_PLATFORM,
    B_HAL_PLATFORM
} platform_type;
```

I haven't tried it, but I really don't think the BeOS would work at all well on a Timex Sinclair (see **is_computer_on_fire()**).

get_system_info() always returns **B_OK**.

---

is_computer_on()

int32 **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.

---

is_computer_on_fire()

double **is_computer_on_fire(**void**)**

Returns the temperature of the motherboard if the computer is currently on fire. Smoldering doesn't count. If the computer isn't on fire, the function returns some other value.

Time Functions

---

real_time_clock() , real_time_clock_usecs() , set_real_time_clock()

uint32 **real_time_clock (**void**)**

```
bigtime_t real_time_clock_usecs (void)
void set_real_time_clock (int32 secs_since_jan1_1970)
```

real_time_clock() returns the number of seconds that have elapsed since January 1, 1970.

real_time_clock_usecs() measures the same time span in microseconds.

set_real_time_clock() sets the value that the other two functions refer to.

---

system_time()

```
bigtime_t system_time(void)
```

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

Thread and Team Concepts

A thread is a synchronous process that executes a series of program instructions. A team is a group of threads that make up a single program or application.

Every application has at least one thread: When you launch an application, an initial threadthe *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 *multithreaded*: 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.

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 "Death and the Main Thread".)

Threads and the POSIX fork() function are not compatible. You can't mix calls to **`spawn_thread()`** (the function that creates a new thread) and **`fork()`** in the same application: If you call **`spawn_thread()`** and then try to call **`fork()`**, the **`fork()`** call will fail. And vice versa.

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.

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_OK)
   /* 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.

Threads and App Images

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_image()`** function. Loading an image causes a separate program, identified as a file, to be launched by the system. For more information on the **`load_image()`** function, see Images.

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()`** is typically used to start the thread that was created through **`load_image()`**.

The Thread Function

When you call **`spawn_thread()`**, you must identify the new thread's *thread 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. The thread function, defined as **`thread_func`**, takes a single (**`void *`**) argument and returns an **`int32`** error code. When the thread function exits, the thread is automatically killed.

You pass a thread function as the first argument to **`spawn_thread()`**. For example, here we spawn a thread that uses a function called **`lister()`** as its thread function. The last argument to spawn_thread() is forwarded to the thread function:

```
int32 lister(void *data)
{
   /* Cast the argument. */
```

```
    BList *listObj = (BList *)data;
    ...
}

int32 main()
{
    BList *listObj = new BList();
    thread_id my_thread;

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

See Passing Data to a Thread for other methods of passing data to a thread.

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 uniquemore 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 Priorities

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" and "real−time."

- **Time−sharing (*values from 1 to 99*)**. A time−sharing thread 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.
- **Real−time (*100 and greater*)**. 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 (see Thread Priority Values for the list). Although you can use other, "in−between" value as the priority argument to **`spawn_thread()`**, it's suggested that you stick with these.

Furthermore, you can call the **`suggest_thread_priority()`** function to let the Kernel Kit determine a good priority for your thread. This function takes information about the thread's scheduling and CPU needs, and returns a reasonable priority value to use when spawning the thread.

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

Semaphores

.

- Synchronization is sometimes a side−effect of sending data between threads. This is explained in "Passing Data to a Thread", and in

Ports

.

- 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 four ways to control a thread while it's running:

- You can put the calling thread to sleep for some number of microseconds through the **`snooze()`** and **`snooze_until()`** functions.
- You can suspend the execution of any thread through the **`suspend_thread()`** function. The thread remains suspended until you "unsuspend" it through a call to **`resume_thread()`** or **`wait_for_thread()`**.
- You can send a POSIX "signal" to a thread through the **`send_signal()`** function. The **`SIGCONT`** signal tries to unblock a blocked or sleeping thread without killing it; all other signals kill the thread. To override this behavior, you can install your own signal handlers.
- You can kill the calling thread through **`exit_thread()`**, or kill some other thread through **`kill_thread()`**. Feeling itchy? 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. However, the death of an application's main thread can affect the other threads:

When a main thread dies, the game is pretty much over. The main thread takes the team's heap, its statically allocated objects, and other team−wide resourcessuch as access to standard IOwith it. This may seriously cripple any threads that linger beyond the death of the main thread.

It's 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

Every thread has a *message cache*. You can write to a thread's message cache through the **`send_data()`** function. The thread can pick up your message (a combination of an integer and a buffer) through **`receive_data()`**. The cache is only one message deep; if there's a message already in the cache, send_data will block. Conversely, if there's no message in the cache, receive_data() will block.

You can also pass data to thread through a port. Arbitrarily deep, ports are more flexible than the message cache. See Ports for details.

Threads and Teams

Declared in: be/kernel/OS.h unless otherwise noted

Library: libroot.so

A thread is a synchronous process that executes a series of program instructions. When you launch an application, an initial threadthe *main thread*is automatically created (or *spawned*) and told to run. 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. The collection of threads that are spawned from the main threadin other words, the threads that comprise an applicationis called a team. All the threads in all teams run concurrently and asynchronously with each other.

For more information on threads and teams, see "Thread and Team Concepts".

Thread and Team Functions

estimate_max_scheduling_latency()

Declared in: be/kernel/scheduler.h

bigtime_t **estimate_max_scheduling_latency(**thread_id *thread* = −1**)**

Returns the scheduling latency, in microseconds, of the specified thread. Specify a **thread_id** of −1 to return the scheduling latency of the current thread.

exit_thread() , kill_thread() , kill_team() , on_exit_thread()

void **exit_thread(**status_t *return_value***)**

status_t **kill_thread(t**hread_id *thread***)**

status_t **kill_team(**team_id *team***)**

status_t **on_exit_thread(**void (**callback*)(void *), void **data***)**

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. **exit_thread()** sends a signal to the thread (after caching the return value in a known place).
- **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_thread()** is the same as sending a **SIGKILLTHR** signal to the thread.
- **kill_team()** kills all the threads within the given team. Again, the threads' return values are random. **kill_team()** is the same as sending a **SIGKILL** signal to any thread in the team. Each of the threads in the team is then handed a **SIGKILLTHR** signal.
- **on_exit_thread()** sets up the specified *callback* to be executed when the calling thread exits. The callback will receive the pointer *data* as an input argument.

Exiting a thread is a fairly safe thing to dosince 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. In addition, killing a thread can leak memory since resources that were allocated by the thread may not be freed. Killing an entire team, on the other hand, won't leak since the system reclaims all resources when the team dies.

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

RETURN CODES

- **B_OK**. The thread or team was successfully killed.
- **B_BAD_THREAD_ID**. Invalid *thread* value.
- **B_BAD_TEAM_ID**. Invalid *team* value.

- **B_NO_MEMORY**. Returned by **on_exit_thread()** if there's no memory to construct the internal callback record.

---

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.

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. There's no way to iterate through identically−named threads.

RETURN CODES

- **B_NAME_NOT_FOUND**. *name* doesn't identify a valid thread.

---

get_team_info() , get_next_team_info()

status_t **get_team_info(team_id** *team*, **team_info** **info***)*

status_t **get_next_team_info(**int32 **cookie*, **team_info** **info***)*

The 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*. For information about the kernel, use **B_SYSTEM_TEAM** as the *team* argument.

The **get_next_team_info()** version lets you step through the list of all teams. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns **B_BAD_VALUE** when there are no more areas to visit:

```
/* Get the team_info for every team. */
team_info info;
int32 cookie = 0;

while (get_next_team_info(0, &cookie, &info) == B_OK)
   ...
```

See **team_info** for a description of that structure.

RETURN CODES

- **B_OK**. The desired team information was found.
- **B_BAD_TEAM_ID**. *team* doesn>t identify an existing team, or there are no more areas to visit.

---

get_thread_info() , get_next_thread_info()

status_t **get_thread_info(**thread_id *thread*, thread_info **info***)*

status_t **get_next_thread_info(**team_id *team*,
int32 *cookie,
thread_info **info***)*

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

The **get_thread_info()** function gets the information for the thread identified by *thread*.

The **get_next_thread_info()** function lets you step through the list of a team's threads through iterated calls. The *team* argument identifies the team you want to look at; a *team* value of 0 means the team of the calling thread. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns **B_BAD_VALUE** when there are no more threads to visit:

```
/* Get the thread_info for every thread in this team. */
thread_info info;
int32 cookie = 0;

while (get_next_thread_info(0, &cookie, &info) == B_OK)
   ...
```

The value of the **priority** field describes the thread's "urgency"; the higher the value, the more urgent the thread. The more urgent the thread, the more attention it gets from the CPU. Expected priority values fall between 0 and 120. See "Thread Priorities" for the full story.

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 validthe thread's state, for example, will almost certainly have changed by the time **get_thread_info()** returns.

RETURN CODES

- **B_OK**. The thread was found; *info* contains valid information.
- **B_BAD_VALUE**. *thread* doesn't identify an existing thread, *team* doesn't identify an existing team, or there are no more threads to visit.

has_data() see send_data()

rename_thread()

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

Changes the name of the given thread to *name*. The name can be no longer than **B_OS_NAME_LENGTH** (32 characters).

RETURN CODES

- **B_OK**. The thread was successfully named.
- **B_BAD_THREAD_ID**. *thread* argument isn't a valid **thread_id** number.

resume_thread()

status_t **resume_thread(**thread_id *thread***)**

Tells a new or suspended thread to begin executing instructions. If the thread has just been spawned, it enters and executes the thread function declared in **spawn_thread()**. If the thread was previously suspended (through **suspend_thread()**), it continues from where it was suspended.

You can't use this function to wake up a sleeping thread, or to unblock a thread that's waiting to acquire a semaphore or waiting in a **receive_data()** call. However, you can unblock any of these threads by suspending and *then* resuming. Blocked threads that are resumed return **B_INTERRUPTED**.

resume_thread() is the same as sending a **SIGCONT** signal to the thread.

RETURN CODES

- **B_OK**. The thread was successfully resumed.
- **B_BAD_THREAD_ID**. *thread* argument isn't a valid **thread_id** number.
- **B_BAD_THREAD_STATE**. The thread isn't suspended.

receive_data()

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 upit 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 ignoredyou 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 that's delivered as **receive_data()**'s return value,
- and an arbitrarily long data buffer that's 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 bufferif the code value returned directly by the function is sufficientyou 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 discardeda 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**).

RETURN CODES

- If successful, the function returns the message's four−byte code.
- **B_INTERRUPTED**. A blocked **receive_data()** call was interrupted by a signal.

---

send_data() , receive_data() , has_data()

status_t **send_data(**thread_id *thread*,
int32 *code*,
void **buffer*,
size_t *buffer_size***)**

int32 **receive_data(**thread_id **sender*,
void **buffer*,
size_t *buffer_size***)**

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

Every thread has a one−message−deep message cache associated with it. These functions access that cache.

send_data() copies a message into *thread*'s message cache. The target thread retrieves the message (and empties the cache) by calling **receive_data()**.

There are two parts to the message:

- A single four−byte *code* passed as an argument to **send_data()** and returned directly by **retrieve_data()**.
- A *buffer* of data that's *buffer_size* bytes long (*buffer* can be **NULL**, in which case *buffer_size* should be 0). The data is copied into the target thread's cache, and then copied into **receive_data()**'s *buffer* (which must be allocated). The calling threads retain responsibility for freeing their buffers.

In addition to returning the *code* directly, and copying the message data into its *buffer* argument, **receive_data()** sets *sender* to the id of the thread that sent the message.

send_data() blocks if there's an unread message in the target thread's cache; otherwise it returns immediately (i.e. it doesn't wait for the target to call **receive_data()**). Analogously, **receive_data()** blocks until there's a message to retrieve.

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

```
main()
{
   thread_id other_thread;
   int32 code = 63;
   char *buf = "Hello";

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

To retrieve the message, the target thread calls **receive_data()**:

```
int32 thread_func(void *data)
{
   thread_id sender;
   int32 code;
   char buf[512];

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

Keep in mind that the message data is *copied* into the buffer; you must allocate adequate storage for the data. If the buffer isn't big enough to accommodate all the datain the message, the left−over portion is thrown away. Note, however, that there isn't any way for a thread to determine how much data has been copied into its message cache.

has_data() returns **true** if *thread* has a message in its message cache. Ostensibly, you use this function before calling **send_data()** or **receive_data()** to avoid blocking:

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

/* or */

if (has_data(find_thread(NULL))
```

```
code = receive_data(...);
```

This works for **receive_data()**, but notice that there's a race condition between the **has_data()** and **send_data()** calls. Another thread could send a message to the target in the interim.

RETURN CODES

send_data() returns:

- **B_OK**. The data was successfuly sent.
- **B_BAD_THREAD_ID**. *thread* doesn't identify a valid thread.
- **B_NO_MEMORY**. The target couldn't allocate enough memory for its copy of *buffer*.
- **B_INTERRUPTED**. The function blocked, but a signal unblocked it.

---

set_thread_priority() , suggest_thread_priority()

status_t **set_thread_priority(**thread_id *thread*, int32 *new_priority***)**

Declared in: be/kernel/scheduler.h

int32 **suggest_thread_priority(**uint32 *what* = B_DEFAULT_MEDIA_PRIORITY,
int32 *period* = 0,
bigtime_t *jitter* = 0,
bigtime_t *length* = 0**)**

set_thread_priority() resets the given thread's priority to *new_priority*. The priority is expected to be between 0 and 120. See "Thread Priorities" for a description of the priority scheme, and "Thread Priority Values" for a list of pre−defined priority constants.

suggest_thread_priority() takes information about a thread and returns a suggested priority that you can pass to **set_thread_priority()** (or, more likely, to **spawn_thread()**).

The *what* value is a bit mask that indicates the type of activities the thread will be used for. The possible values are listed in Suggested Thread Priorities.

period is the number of times per second the thread needs to be run (specify 0 if it needs to run continuously). *jitter* is an estimate, in microseconds, of how much the period can vary as long as the average stays at *period* times per second.

length is an approximation of the amount of time, in microseconds, the thread will typically run per invocation (i.e., the amount of time that will pass between the moment it receives a message, through processing it, until it's again waiting for another message).

For example, if you're spawning a thread to handle video refresh for a computer game, and you want the display to update 30 times per second, you might use code similar to the following:

```
int32 priority;
priority = suggest_thread_priority(B_LIVE_3D_RENDERING, 30, 1000, 150);
th = spawn_thread(func, "render_thread", priority, NULL)
```

This spawns the rendering thread with a priority appropriate for a thread for live 3D rendering which wants to be run 30 times per second, with a variation of only 1000 microseconds. Each invocation of the thread's code is estimated to take 150 microseconds. Obviously the *jitter* and *length* values would have to be tuned to the particular application.

RETURN CODES

set_thread_priority() returns...

- *Positive integers.* If the function is successful, the previous priority is returned.
- **B_BAD_THREAD_ID**. *thread* doesn't identify a valid thread.

---

snooze() , snooze_until()

status_t **snooze(**bigtime_t *microseconds***)**

status_t **snooze_until(**bigtime_t *microseconds*, int *timebase***)**

snooze() blocks the calling thread for the given number of microseconds.

snooze_until() blocks until an absolute time measured in the given timebase. Currently, the only allowed value for timebase is

B_SYSTEM_TIMEBASE, which measures time against the system clock (as reported by **system_time()**).

RETURN CODES

- **B_OK**. The thread went to sleep and is now awake.
- **B_INTERRUPTED**. The thread received a signal while it was sleeping.

---

snooze_until() see snooze()

---

spawn_thread()

```
thread_id spawn_thread(thread_func func,
    const char *name,
    int32 priority,
    void *data)
```

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

- *func* is a pointer to a thread function. This is the function that the thread will execute when it's told to run. See "The Thread Function" for details.
- *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. This value should be between 0 and 120; he higher the priority, the more attention the thread gets. See

---

for adescription of the priorities, and

Thread Priority Values

for a list of priority constants.

- *data* is forwarded as the argument to the thread 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 thread function exits, or until the thread is explicitly killed (through a signal or a call to **exit_thread()**, **kill_thread()**, or **kill_team()**).

RETURN CODES

- **B_NO_MORE_THREADS**. all **thread_id** numbers are currently in use.
- **B_NO_MEMORY**. Not enough memory to allocate the resources for another thread.

---

suggest_thread_priority() see set_thread_priority()

---

suspend_thread()

status_t **suspend_thread(**thread_id *thread***)**

Halts the execution of the given thread, but doesn't kill the thread entirely. The thread remains suspended (**suspend_thread()** blocks) until it's 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.

You can suspend any thread, regardless of its current state. But be careful: If the thread is blocked on a semaphore (for example), the subsequent **resume_thread()** call will "hop over" the semaphore acquisition.

Suspensions don't nest. A single **resume_thread()** unsuspends a thread regardless of the number of **suspend_thread()** calls it has received.

suspend_thread() is the same as sending a **SIGSTOP** signal to the thread.

RETURN CODES

- **B_OK**. The thread is now suspended.
- **B_BAD_THREAD_ID**. *thread* isn't a valid **thread_id** number.

---

wait_for_thread()

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

This function causes the calling thread to wait until *thread* (the "target thread") has died. If *thread* is suspended (or freshly spawned), **wait_for_thread()** will resume it.

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

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.

RETURN CODES

- **B_OK**. The target is now dead.
- **B_BAD_THREAD_ID**. *thread* isn't a valid **thread_id** number.
- **B_INTERRUPTED**. The target was killed by a signal. This includes **kill_thread()**, **kill_team()**, and **exit_thread()**.

Thread and Team Structures and Types

team_id , thread_id

typedef int32 **team_id** ;

typedef int32 **thread_id** ;

These id numbers uniquely identify teams and threads, respecitvely.

team_info

```
typedef struct {
    team_id team;
    int32 thread_count;
    int32 image_count;
    int32 area_count;
        thread_id debugger_nub_thread;
        port_id debugger_nub_port;
    int32 argc;
    char args[64];
    uid_t uid;
    gid_t gid;
    } team_info;
```

The **team_info** structure returns information about a team. To retrieve one of these structures, use **get_team_info()** or **get_next_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.

uid and **gid** identify the user and group that "owns" the team. You can use these values to play permission games.

thread_func

```
typedef int32 (*thread_func)(void *data);
```

thread_func is the prototype for a thread's *thread function*. You specify a thread function by passing a **thread_func** as the first argument to **spawn_thread()**; the last argument to **spawn_thread()** is forwarded as the thread function's *data* argument. When the thread function exits, the spawned thread is automatically killed. To retrieve a **thread_func**'s return value, some other thread must be waiting in a **wait_for_thread()** call.

Note that **spawn_thread()** *doesn't* copy the data that *data* points to. It simply passes the pointer through literally. Never pass a pointer that's allocated locally (on the stack).

thread_info

```
typedef struct {
        thread_id thread;
        team_id team;
    char name[B_OS_NAME_LENGTH];
        thread_state state;
        sem_id sem;
    int32 priority;
        bigtime_t user_time;
        bigtime_t kernel_time;
    void *stack_base;
    void *stack_end;
    } thread_info
```

The thread_info structure contains information about a thread. To retrieve one of these structure, use **get_thread_info()** or

get_next_thread_info().

The **thread**, **team**, and **name** fields contain the indicated information.

state describes what the thread is currently doing (see **thread_state** for the list of states). If the thread is waiting to acquire a semaphore, **sem** is that semaphore.

priority is a value that indicates the level of attention the thread gets (see Thread Priority).

user_time and **kernel_time** are the amounts of time, in microseconds, the thread has spent executing user code and the amount of time the kernel has run on the thread's behalf, respectively.

stack_base and **stack_end** are pointers to the first byte and last bytes in the thread's execution stack. Currently, the stack size is fixed at around 256k.

The two stack pointers are currently inverted such that **stack_base** is *less* than **stack_end**. (In a stack−grows−down world, the base should be greater than the end.)

Thread and Team Constants

B_SYSTEM_TEAM

```
#define B_SYSTEM_TEAM ...
```

Use this constant as the first argument to **get_team_info()** to get team information about the kernel).

B_SYSTEM_TIMEBASE

```
#define B_SYSTEM_TIMEBASE ...
```

The system timebase constant is used as a basis for time measurement in the **snooze_until()** function. (Currently, it's the only timebase available.)

be_task_flags

Declared in: be/kernel/scheduler.h

```
enum be_task_flags { B_DEFAULT_MEDIA_PRIORITY,
        B_OFFLINE_PROCESSING,
        B_STATUS_RENDERING,
        B_USER_INPUT_HANDLING
          B_LIVE_VIDEO_MANIPULATION
    };
```

B_DEFAULT_MEDIA_PRIORITY	The thread isn't doing anything specialized.
B_OFFLINE_PROCESSING	The thread is doing non−real−time computations.
B_STATUS_RENDERING	The thread is rendering a status or preview display.
B_USER_INPUT_HANDLING	The thread is handling user input.
B_LIVE_VIDEO_MANIPULATIO N	The thread is processing live video (filtering, compression, decompression, etc.).
B_VIDEO_PLAYBACK	The thread is playing back video from a hardware device.

B_VIDEO_RECORDING	The thread is recording video from a hardware device.
B_LIVE_AUDIO_MANIPULATIO N	The thread is doing real−time manipulation of live audio data (filtering, compression, decompression, etc.).
B_AUDIO_PLAYBACK	The thread is playing back audio from a hardware device.
B_AUDIO_RECORDING	The thread is recording audio from a hardware device.
B_LIVE_3D_RENDERING	The thread is performing live 3D rendering.
B_NUMBER_CRUNCHING	The thread is doing data processing.

These constants describe what the thread is designed to do. You use these constants when asking for a suggested priority (see **suggest_thread_priority()**).

These constants may not be used as actual thread priority valuesdo not pass one of these values as the *priority* argument to **spawn_thread()**.

Thread Priority Values

B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORIT Y	20

B_REAL_TIME_DISPLAY_PRIORITY	**100**
B_URGENT_PRIORITY	**110**
B_REAL_TIME_PRIORITY	**120**

The thread priority values are used to set the "urgency" of a thread. Although you can reset a thread's priority through **set_thread_priority()**, the priority is initiallyand almost always permanentlyset in **spawn_thread()**. As shown here, there are two types of

thread_state

enum { ... } **thread_state**

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. The **sem** field of the thread's **thread_info** structure will tell you which semaphore.
B_THREAD_RECEIVING	The thread is sitting in a **receive_data()** function call.

B_THREAD_ASLEEP	The thread is sitting in a **snooze()** call.

A thread's state tells you what the thread is currently doing. To get the state, look in the **state** field of the **thread_info** structure (retrieved through **get_thread_info()**).

Time Information

Declared in: be/kernel/OS.h

Library: libroot.so

The following functions set and get the system clock.

Time Functions

real_time_clock() , real_time_clock_usecs() , set_real_time_clock()

```
uint32 real_time_clock (void)
bigtime_t real_time_clock_usecs (void)
void set_real_time_clock (int32 secs_since_jan1_1970)
```

`real_time_clock()` returns the number of seconds that have elapsed since January 1, 1970.

`real_time_clock_usecs()` measures the same time span in microseconds.

`set_real_time_clock()` sets the value that the other two functions refer to.

system_time()

```
bigtime_t system_time(void)
```

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

Miscellaneous Functions and Constants

Declared in: be/kernel/OS.h (unless otherwise noted)

Functions

clear_caches()

Declared in: be/kernel/image.h

```
void clear_caches(void *addr, size_t len, uint32 flags)
```

This function clears or invalidates the instruction and data caches. You should only need this function if you're generating code on the fly, or if you're performing a timing loop and you want to start with fresh caches (to get a "worst case" estimate).

The argument are:

- *addr* is the starting address of a section of memory that corresponds to a section of one of the caches.
- *len* is the length, in bytes, of the instruction or data segment that you want to clear or invalidate.
- *flags* is one or both of **B_INVALIDATE_ICACHE** and **B_FLUSH_DCACHE**.

By invalidating a section of the instruction cache, you cause the instructions in that section to be reloaded next time they're needed. Flushing the data cache causes the in−memory copy of the data to be written out to the cache.

debugger()

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

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

disable_debugger()

```
int disable_debugger(int state)
```

Instructs the kernel to send a signal for all exceptions, even those that don't normally trigger the debugger. If the application doesn't have a handler installed for the exception, the team dies without triggering the debugger. *state* should be nonzero to turn on this functionality or 0 to turn it off.

set_alarm()

```
bigtime_t set_alarm(bigtime_t time, uint32 mode)
```

Tells the kernel to send the **SIGALRM** signal at some point in the future, as defined by the arguments:

- If *mode* is **B_PERIODIC_ALARM**, the signal is sent every *time* microseconds, starting as soon as **set_alarm()** function returns.
- If *mode* is **B_ONE_SHOT_ABOLUTE_ALARM**, the signal is sent once (only) after *time* microseconds have elapsed measured from the time the system was booted. If that point has already passed, the signal is sent immediately.
- If *mode* is **B_ONE_SHOT_RELATIVE_ALARM**, the signal is sent once (only) after *time* microseconds have elapsed from the time **set_alarm()** returns.

When the signal is sent, the **SIGALRM** handler is called (you set the handler through the normal means, by calling the Posix **signal()** function). The handler runs in the thread that set the alarm.

> From within the **SIGALRM** handler, you mustn't call anything that would cause the kernel scheduler to run. Just about the only safe call you can make from your signal handler is **release_sem()**.

The most recent alarm requested cancels any previous request. For example, in this sequence...

```
/* Ask for an alarm ten seconds from now. */
set_alarm(10e6, B_ONE_SHOT_RELATIVE_ALARM);
/* Ask for an alarm one second from now. */
set_alarm(10e5, B_ONE_SHOT_RELATIVE_ALARM);
```

...only the second alarm request will be fulfilledthe first requested is cancelled when the second **`set_alarm()`** call is made. This applies to all alarm types; for example, a one−shot alarm request will cancel an active periodic alarm.

To explicitly cancel the previous alarm request without installing a new alarm, do this:

```
set_alarm(B_INFINITE_TIMEOUT, B_PERIODIC_ALARM);
```

This cancels the previous alarm request regardless of the type of alarm.

set_signal_stack()

Declared in: posix/signal.be.h

void **`set_signal_stack(`**void *_ptr_, size_t _size_**`)`**

Sets the location and size of the stack that's used by the thread's signal handlers.

Constants

B_INFINITE_TIMEOUT

`B_INFINITE_TIMEOUT`

The inifinite timeout value can be used to specify, to timeout−accepting functions, that you're willing to wait forever.

B_OS_NAME_LENGTH

`B_OS_NAME_LENGTH`

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

B_PAGE_SIZE

`B_PAGE_SIZE`

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

The Kernel Kit: Master Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W