4

Mach Functions

This chapter gives detailed descriptions of the C functions provided by the NeXT Mach operating system.   It also describes some macros that behave like functions.   For this chapter, the functions and macros are divided into five groups:

· C-thread functionsÐUse these to implement multiple threads in an application.

· Mach kernel functionsÐUse these to get access to the Mach operating system.

· Bootstrap Server functionsÐUse these to set up communication between the task that provides a local service and the tasks that use the service.

· Network Name Server functionsÐUse these to set up communication between tasks that might not be on the same machine.

· Kernel-server loader functionsÐUse these to load and unload loadable kernel servers, to add and delete servers to and from the kernel-server loader, and to get information about servers.

Within each section, functions are subgrouped with other functions that perform related tasks.   These subgroups are described in alphabetical order by the name of the first function listed in the subgroup.   Functions within subgroups are also listed alphabetically, with a pointer to the subgroup description.

For convenience, these functions are summarized in the *NEXTSTEP Programming Interface Summary*.   The summary lists functions by the same subgroups used in this chapter and combines several related subgroups under a heading such as ªBasic C-Thread€Functionsº or ªTask Functions.º   For each function, the summary shows the calling sequence.

C-Thread Functions

These functions provide a C language interface to the low-level, language-independent primitives for manipulating threads of control.

In a multithreaded application, you should use the C-thread functions whenever possible, rather than Mach kernel functions.   If you need to call a Mach kernel function that requires a **thread_t** argument, you can find the Mach thread that corresponds to a particular C thread by calling **cthread_thread()**.

condition_alloc(), mutex_alloc()

SUMMARY Create a condition or mutex object

SYNOPSIS **#import <mach/cthreads.h>**

condition_t **condition_alloc(**void**)**
mutex_t **mutex_alloc(**void**)**

DESCRIPTION The macros **condition_alloc()** and **mutex_alloc()** provide dynamic allocation of condition and mutex objects. When you're finished using these objects, you can deallocate them using **condition_free()** and **mutex_free()**.

EXAMPLE

```
my_condition = condition_alloc();
my_mutex = mutex_alloc();
```

SEE ALSO **condition_init()**, **mutex_init()**, **condition_free()**, **mutex_free()**

condition_broadcast()

SUMMARY Broadcast a condition

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_broadcast(**condition_t *c***)**

DESCRIPTION The macro **condition_broadcast()** wakes up all threads that are waiting (with **condition_wait()**) for the condition *c*. This macro is similar to **condition_signal()**, except that **condition_signal()** doesn't wake up every waiting thread.

EXAMPLE

```
any_t listen(any_t arg)
{
    mutex_lock(my_mutex);
    while(!data)
        condition_wait(my_condition, my_mutex);
    /* . . . */
    mutex_unlock(my_mutex);

    mutex_lock(printing);
    printf("Condition has been met\n");
    mutex_unlock(printing);
}

main()
{
    my_condition = condition_alloc();
    my_mutex = mutex_alloc();
    printing = mutex_alloc();

    cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)0));

    mutex_lock(my_mutex);
    data = 1;
    mutex_unlock(my_mutex);
    condition_broadcast(my_condition);
    /* . . . */
}
```

SEE ALSO **condition_signal()**, **condition_wait()**

condition_clear(), mutex_clear()

SUMMARY Clear a condition or mutex object

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_clear(**struct condition **c***)**
void **mutex_clear(**struct mutex **m***)**

DESCRIPTION You must call one of these macros before freeing an object of type **struct condition** or **struct mutex**. See the discussion of **condition_init()** and **mutex_init()** for information on why you might want to use these types instead of **condition_t** and **mutex_t**.

EXAMPLE
```
struct mystruct {
    my_data_t     data;
    struct mutex  m;
};
struct mystruct  *mydata;
mydata = (struct mystruct *)malloc(sizeof (struct mystruct));

mutex_init(&mydata->m);
/* . . . */
mutex_lock(&mydata->m);
/* Do something to mydata that only one thread can do. */
mutex_unlock(&mydata->m);
/* . . . */
mutex_clear(&mydata->m);
free(mydata);
```

SEE ALSO **condition_init()**, **mutex_init()**, **condition_free()**, **mutex_free()**

condition_free(), mutex_free()

SUMMARY Deallocate a condition or mutex object

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_free(**condition_t *c***)**
void **mutex_free(**mutex_t *m***)**

DESCRIPTION The macros **condition_free()** and **mutex_free()** let you deallocate condition and mutex objects that were allocated dynamically. Before deallocating such an object, you must guarantee that no other thread will reference it. In particular, a thread blocked in **mutex_lock()** or **condition_wait()** should be viewed as referencing the object continually; freeing the object out from under such a thread is erroneous, and can result in bugs that are extremely difficult to track down.

SEE ALSO **condition_alloc()**, **mutex_alloc()**, **condition_clear()**, **mutex_clear()**

condition_init(), mutex_init()

SUMMARY Initialize a condition variable or mutex

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_init(**struct condition **c***)**
void **mutex_init(**struct mutex **m***)**

DESCRIPTION The macros **condition_init()** and **mutex_init()** initialize an object of the **struct condition** or **struct mutex** referent type, so that its address can be used wherever an object of type **condition_t** or **mutex_t** is expected. Initialization of the referent type is most often used when you have included the referent type itself (rather than a pointer) in a larger structure, for more efficient storage allocation.

For instance, a data structure might contain a component of type **struct mutex** to allow each instance of that structure to be locked independently. During initialization of the instance, you would call **mutex_init()** on the **struct mutex** component. The alternative of using a **mutex_t** component and initializing it using **mutex_alloc()** would be less efficient.

If you're going to free a condition or mutex object of type **struct condition** or **struct mutex**, you should first clear it using **condition_clear()** or **mutex_clear()**.

EXAMPLE

```
struct mystruct {
    my_data_t     data;
    struct mutex  m;
};
struct mystruct  *mydata;
mydata = (struct mystruct *)malloc(sizeof (struct mystruct));

mutex_init(&mydata->m);
/* . . . */
mutex_lock(&mydata->m);
/* Do something to mydata that only one thread can do. */
mutex_unlock(&mydata->m);
/* . . . */
mutex_clear(&mydata->m);
free(mydata);
```

SEE ALSO **condition_alloc()**, **mutex_alloc()**, **condition_clear()**, **mutex_clear()**

condition_name(), condition_set_name(), mutex_name(), mutex_set_name()

SUMMARY Associate a string with a condition or mutex variable

SYNOPSIS **#import <mach/cthreads.h>**

char ***condition_name(**condition_t *c***)**
void **condition_set_name(**condition_t *c*, char **name***)**
char ***mutex_name(**mutex_t *m***)**
void **mutex_set_name(**mutex_t *m*, char **name***)**

DESCRIPTION These macros let you associate a name with a condition or a mutex object. The name is used when trace information is displayed. You can also use this name for your own application-dependent purposes.

EXAMPLE

```
/* Do something if this is a "TYPE 1" condition. */
if (strcmp(condition_name(c), "TYPE 1") == 0)
    /* Do something. */;
```

condition_signal()

SUMMARY Signal a condition

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_signal(**condition_t *c***)**

DESCRIPTION The macro **condition_signal()** should be called when one thread needs to indicate that the condition represented by the condition variable is now true. If any other threads are waiting (using **condition_wait()**), at least one of them will be awakened. If no threads are waiting, nothing happens. The macro **condition_broadcast()** is similar to this one, except that it wakes up *all* threads that are waiting.

EXAMPLE

```
any_t listen(any_t arg)
{
        mutex_lock(my_mutex);
        while(!data)
            condition_wait(my_condition, my_mutex);
        /* . . . */
        mutex_unlock(my_mutex);

        mutex_lock(printing);
        printf("Condition has been met\n");
        mutex_unlock(printing);
    }

    main()
    {
        my_condition = condition_alloc();
        my_mutex = mutex_alloc();
        printing = mutex_alloc();

        cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)0));

        mutex_lock(my_mutex);
        data = 1;
        mutex_unlock(my_mutex);
        condition_signal(my_condition);
        /* . . . */
    }
```

SEE ALSO **condition_broadcast()**, **condition_wait()**

condition_wait()

SUMMARY Wait on a condition

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_wait(**condition_t *c*, mutex_t *m***)**

DESCRIPTION The function **condition_wait()** unlocks the mutex it takes as a argument, suspends the calling thread until the specified condition is likely to be true, and locks the mutex again when the thread resumes. There's no guarantee that the condition will be true when the thread resumes, so this function should always be used as follows:

```
mutex_t m;
condition_t c;

mutex_lock(m);
/* . . . */
while    (/* condition isn't true */)
    condition_wait(c, m);
/* . . . */
mutex_unlock(m);
```

SEE ALSO **condition_broadcast()**, **condition_signal()**

cthread_abort()

SUMMARY Interrupt a C thread

SYNOPSIS **#import <mach/cthreads.h>**

kern_return_t **cthread_abort(**cthread_t *t***)**

DESCRIPTION This function provides the functionality of **thread_abort()** to C threads. The **cthread_abort()** function interrupts system calls; it's usually used along with **thread_suspend()**, which stops a thread from executing any more user code. Calling **cthread_abort()** on a thread that isn't suspended is risky, since it's difficult to know exactly what system trap, if any, the thread might be executing and whether an interrupt return would cause the thread to do something useful.

See **thread_abort()** for a full description of the use of this function.

cthread_count()

SUMMARY Get the number of threads in this task

SYNOPSIS **#import <mach/cthreads.h>**

int **cthread_count()**

DESCRIPTION This function returns the number of threads that exist in the current task. You can use this function to help make sure that your task doesn't create too many threads (over 200 or so). See **cthread_set_limit()** for information on restricting the number of threads in a task.

EXAMPLE
```
printf("C thread count should be 1, is %d\n", cthread_count());
cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)0));
printf("C thread count should be 2, is %d\n", cthread_count());
```

SEE ALSO **cthread_limit()**, **cthread_set_limit()**

cthread_data(), cthread_set_data()

SUMMARY Associate data with a thread

SYNOPSIS **#import <mach/cthreads.h>**

any_t **cthread_data(**cthread_t *t***)**
void **cthread_set_data(**cthread_t *t*, any_t *data***)**

DESCRIPTION The macros **cthread_data()** and **cthread_set_data()** let you associate arbitrary data with€a€thread, providing a simple form of thread-specific ªglobalº variable. More elaborate€mechanisms, such as per-thread property lists or hash tables, can then be built with these macros.

EXAMPLE
```
int listen(any_t arg)
{
    mutex_lock(printing);
    printf("This thread's data is: %d\n",
        (int)cthread_data(cthread_self()));
    mutex_unlock(printing);
    /* . . . */
}

main()
{
    cthread_t lthread;

    printing = mutex_alloc();

    lthread = cthread_fork((cthread_fn_t)listen, (any_t)0);
    cthread_set_data(lthread, (any_t)100);
    cthread_detach(lthread);
    /* . . . */
}
```

SEE ALSO **cthread_name()**, **cthread_set_name()**

cthread_detach()

SUMMARY Detach a thread

SYNOPSIS **#import <mach/cthreads.h>**

void **cthread_detach(**cthread_t *t***)**

DESCRIPTION The function **cthread_detach()** is used to indicate that **cthread_join()** will never be called on the given thread. This is usually known at the time the thread is forked, so the most efficient usage is the following:

cthread_detach(cthread_fork(*function*, *argument***))**;

A thread may, however, be detached at any time after it's forked, as long as no other attempt is made to join it or detach it.

EXAMPLE

```
cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)reply_port));
```

SEE ALSO **cthread_fork()**, **cthread_join()**

cthread_errno()

SUMMARY Get a thread's errno value

SYNOPSIS **#import <mach/cthreads.h>**

int **cthread_errno(**void**)**

DESCRIPTION Use the **cthread_errno()** function to get the errno value for the current thread. In the UNIX operating system, **errno** is a process-wide global variable that's set to an error number when a UNIX system call fails. However, because Mach has multiple threads per process, Mach keeps errno information on a per-thread basis as well as in **errno**.

Like the value of **errno**, the value returned by **cthread_errno()** is valid only if the last UNIX system call returned -1. Errno values are defined in the header file **bsd/sys/errno.h**.

EXAMPLE

```
int ret;

ret = chown(FILEPATH, newOwner, newGroup);
if (ret == -1) {
    if (cthread_errno() == ENAMETOOLONG)
        /* . . . */
}
```

SEE ALSO **cthread_set_errno()**, **intro(2)** UNIX manual page

cthread_exit()

SUMMARY Exit a thread

SYNOPSIS **#import <mach/cthreads.h>**

void **cthread_exit(**any_t *result***)**

DESCRIPTION The function **cthread_exit()** terminates the calling thread. The result is passed to the thread that joins the caller, or is discarded if the caller is detached.

An implicit **cthread_exit()** occurs when the top-level function of a thread returns, but it may also be called explicitly.

EXAMPLE

```
cthread_exit(0);
```

SEE ALSO **cthread_detach()**, **cthread_fork()**, **cthread_join()**

cthread_fork()

SUMMARY Fork a thread

SYNOPSIS **#import <mach/cthreads.h>**

cthread_t **cthread_fork(**any_t (**function*)(), any_t *arg***)**

DESCRIPTION The function **cthread_fork()** takes two arguments: a function for the new thread to execute, and an argument to this function. The **cthread_fork()** function creates a new thread of control in which the specified function is executed concurrently with the caller's thread. This is the sole means of creating new threads.

The **any_t** type represents a pointer to any C type. The **cthread_t** type is an integer-size handle that uniquely identifies a thread of control. Values of type **cthread_t** will be referred to as thread identifiers. Arguments larger than a pointer must be passed by reference. Similarly, multiple arguments must be simulated by passing a pointer to a structure containing several components. The call to **cthread_fork()** returns a thread identifier that can be passed to **cthread_join()** or **cthread_detach()**. Every thread must be either joined or detached exactly once.

EXAMPLE

```
cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)reply_port));
```

SEE ALSO **cthread_detach()**, **cthread_exit()**, **cthread_join()**

cthread_join()

SUMMARY Join threads

SYNOPSIS **#import <mach/cthreads.h>**

any_t **cthread_join(**cthread_t *t***)**

DESCRIPTION The function **cthread_join()** suspends the caller until the specified thread *t* terminates. The caller receives either the result of *t*'s top-level function or the argument with which *t* explicitly called **cthread_exit()**.

Attempting to join one's own thread results in deadlock.

EXAMPLE

```
cthread_t t;
t = cthread_fork((any_t (*)())listen, (any_t)reply_port);
/* . . . (Do some work, perhaps forking other threads.) */
result = cthread_join(t);  /* Wait for the thread to finish executing. */
/* . . . (Continue doing work) */
```

SEE ALSO **cthread_detach()**, **cthread_exit()**, **cthread_fork()**

cthread_limit(), cthread_set_limit()

SUMMARY Get or set the maximum number of threads in this task

SYNOPSIS **#import <mach/cthreads.h>**

int **cthread_limit(**void**)**
void **cthread_set_limit(**int *limit***)**

ARGUMENTS *limit*: The new maximum number of C threads per task. Specify zero if you want no limit.

DESCRIPTION These functions can help you to avoid creating too many threads. The danger in creating a large number of threads is that the kernel might run out of resources and panic. Usually, a task should avoid creating more than about 200 threads.

Use **cthread_set_limit()** to set a limit on the number of threads in the current task. When the limit is reached, new C threads will appear to fork successfully. However, they will have no associated Mach thread, so they won't do anything.

Use **cthread_limit()** to find out how many threads can exist in the current task. If the returned value is zero (the default), then no limit is currently being enforced.

Important: Use **cthread_count()** to determine when your task is approaching the maximum number of threads.

EXAMPLE

```
cthread_set_limit(LIMIT);

/* . . . */

/* Fork if we haven't reached the limit. */
if ( (LIMIT == 0) || (LIMIT > cthread_count()) )
    cthread_detach(cthread_fork((any_t (*)())a_thread,(any_t)0));
```

cthread_name(), cthread_set_name()

SUMMARY Associate a string with a thread

SYNOPSIS **#import <mach/cthreads.h>**

char ***cthread_name(**cthread_t *t***)**
void **cthread_set_name(**cthread_t *t*, char **name***)**

DESCRIPTION The functions **cthread_name()** and **cthread_set_name()** let you associate an arbitrary name with a thread. The name is used when trace information is displayed. The name may also be used for application-specific diagnostics.

EXAMPLE

```
int listen(any_t arg)
{
    mutex_lock(printing);
    printf("This thread's name is: %s\n",
        cthread_name(cthread_self()));
    mutex_unlock(printing);
    /* . . . */
}

main()
{
    cthread_t lthread;

    printing = mutex_alloc();

    lthread = cthread_fork((cthread_fn_t)listen, (any_t)0);
    cthread_set_name(lthread, "lthread");
    cthread_detach(lthread);
    /* . . . */
}
```

SEE ALSO **cthread_data()**, **cthread_set_data()**

cthread_priority(), cthread_max_priority()

SUMMARY Set the scheduling priority for a C thread

SYNOPSIS **#import <mach/cthreads.h>**

kern_return_t **cthread_priority(**cthread_t *t*, int *priority*, boolean_t *set_max***)**
kern_return_t **cthread_max_priority(**cthread_t *t*, processor_set_t *processor_set*, int€*max_priority***)**

ARGUMENTS *t*: The C thread whose priority is to be changed.

priority: The new priority to change it to.

set_max: Also set *t*'s maximum priority if true.

processor_set: The privileged port for the processor set to which *thread* is currently assigned.

max_priority: The new maximum priority.

DESCRIPTION These functions give C threads the functionality of **thread_priority()** and **thread_max_priority()**. See those functions for more details than are provided here.

The **cthread_priority()** function changes the base priority and (optionally) the maximum priority of *t*. If the new base priority is higher than the scheduled priority of the currently executing thread, this thread might be preempted. The maximum priority of the thread is also set if *set_max* is true. This call fails if *priority* is greater than the current maximum priority of the thread. As a result, **cthread_priority()** can lowerÐbut never raiseÐthe value of a thread's maximum priority.

The **cthread_max_priority()** function changes the maximum priority of the thread. Because it requires the privileged port for the processor set, this call can reset the maximum priority to any legal value. If the new maximum priority is less than the thread's base priority, then the thread's base priority is set to the new maximum priority.

EXAMPLE

```
/* Get the privileged port for the default processor set. */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error!=KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

/* Set the max priority. */
error=cthread_max_priority(cthread_self(), default_set_priv,
    priority);
if (error!=KERN_SUCCESS)
    mach_error("Call to cthread_max_priority() failed",error);

/* Set the thread's priority. */
error=cthread_priority(cthread_self(), priority, FALSE);
```

```
if (error!=KERN_SUCCESS)
    mach_error("Call to cthread_priority() failed",error);
```

RETURN KERN_SUCCESS: Operation completed successfully

KERN_INVALID_ARGUMENT: *cthread* is not a C thread, *processor_set* is not a privileged port for a processor set, or *priority* is out of range (not in 0-31).

KERN_FAILURE: The requested operation would violate the thread's maximum priority (only for **cthread_priority()**) or the thread is not assigned to the processor set whose privileged port was presented.

SEE ALSO **thread_priority()**, **thread_max_priority()**, **thread_policy()**, **task_priority()**, **processor_set_priority()**

cthread_self()

SUMMARY Return the caller's C-thread identifier

SYNOPSIS **#import <mach/cthreads.h>**

cthread_t **cthread_self(**void**)**

DESCRIPTION The function **cthread_self()** returns the caller's own C-thread identifier, which is the same value that was returned by **cthread_fork()** to the creator of the thread. The C-thread identifier uniquely identifies the thread, and hence may be used as a key in data structures that associate user data with individual threads. Since thread identifiers may be reused by the underlying implementation, you should be careful to clean up such associations when threads exit.

EXAMPLE
```
printf("This thread's name is: %s\n",
    cthread_name(cthread_self()));
mutex_unlock(printing);
```

SEE ALSO **cthread_fork()**, **cthread_thread()**, **thread_self()**

cthread_set_errno_self()

SUMMARY Set the current thread's errno value

SYNOPSIS **#import <mach/cthreads.h>**

void **cthread_set_errno_self(**int *error***)**

DESCRIPTION Use this function to set the errno value for the current thread to *error*. In the UNIX operating system, **errno** is a process-wide global variable that's set to an error number when a UNIX system call fails. However, because Mach has multiple threads per process, Mach keeps errno information on a per-thread basis as well as in **errno**. This function has no effect on the value of **errno**.

The current thread's errno value can be obtained by calling **cthread_errno()**. Errno values are defined in the header file **bsd/sys/errno.h**.

EXAMPLE
```
cthread_set_errno_self(EPERM);
```

SEE ALSO **cthread_errno()**, **intro(2)** UNIX manual page

cthread_thread()

SUMMARY Return the caller's Mach thread identifier

SYNOPSIS **#import <mach/cthreads.h>**

thread_t **cthread_thread(**cthread_t *t***)**

DESCRIPTION The macro **cthread_thread()** returns the Mach thread that corresponds to the specified C thread *t*.

EXAMPLE

```
/* Save the cthread and thread values for the forked thread. */
l_cthread = cthread_fork((cthread_fn_t)listen, (any_t)0);
cthread_detach(l_cthread);
l_realthread = cthread_thread(l_cthread);
```

SEE ALSO **cthread_fork()**, **cthread_self()**

cthread_yield()

SUMMARY Yield the processor to other threads

SYNOPSIS **#import <mach/cthreads.h>**

void **cthread_yield(**void**)**

DESCRIPTION The function **cthread_yield()** is a hint to the scheduler, suggesting that this would be a convenient point to schedule another thread to run on the current processor.

EXAMPLE

```
int i, n;

/* n is set previously */
for (i = 0; i < n; i += 1)
    cthread_yield();
```

SEE ALSO **cthread_priority()**, **thread_switch()**

mutex_lock()

SUMMARY Lock a mutex variable

SYNOPSIS **#import <mach/cthreads.h>**

void **mutex_lock(**mutex_t *m***)**

DESCRIPTION The macro **mutex_lock()** attempts to lock the mutex *m* and blocks until it succeeds. If several threads attempt to lock the same mutex concurrently, one will succeed, and the others will block until *m* is unlocked. A deadlock occurs if a thread attempts to lock a mutex it has already locked.

EXAMPLE

```
/* Only one thread at a time should call printf. */
mutex_lock(printing);
printf("Condition has been met\n");
mutex_unlock(printing);
```

SEE ALSO **mutex_try_lock()**, **mutex_unlock()**

mutex_try_lock()

SUMMARY Try to lock a mutex variable

SYNOPSIS **#import <mach/cthreads.h>**

int **mutex_try_lock(**mutex_t *m***)**

DESCRIPTION The function **mutex_try_lock()** attempts to lock the mutex *m*, like **mutex_lock()**, and returns true if it succeeds. If *m* is already locked, however, **mutex_try_lock()** immediately returns false rather than blocking. For example, a busy-waiting version of **mutex_lock()** could be written using **mutex_try_lock()**:

```
void mutex_lock(mutex_t m)
{
    for (;;)
        if (mutex_try_lock(m))
            return;
}
```

SEE ALSO **mutex_lock()**, **mutex_unlock()**

mutex_unlock()

SUMMARY Unlock a mutex variable

SYNOPSIS **#import <mach/cthreads.h>**

void **mutex_unlock(**mutex_t *m***)**

DESCRIPTION The function **mutex_unlock()** unlocks *m*, giving other threads a chance to lock it.

EXAMPLE

```
/* Only one thread at a time should call printf. */
mutex_lock(printing);
printf("Condition has been met\n");
mutex_unlock(printing);
```

SEE ALSO **mutex_lock()**, **mutex_try_lock()**

Mach Kernel Functions

exc_server()

SUMMARY Dispatch a message received on an exception port

SYNOPSIS **#import <mach/mach.h>**
#import <mach/exception.h>

boolean_t **exc_server(**msg_header_t **in*, msg_header_t **out***)**

ARGUMENTS *in*: A message that was received on the exception port. This message structure should be at least 64 bytes long.

out: An empty message to be filled by **exc_server()** and then sent. This message buffer should be at least 32 bytes long.

DESCRIPTION This function calls the appropriate exception handler. You should call this function after you've received a message on an exception port that you set up previously. Usually, this function is used along with a user-defined exception handler, which must have the following protocol:

kern_return_t **catch_exception_raise(**port_t *exception_port*, port_t *thread*, port_t€*task*, int€*exception*, int€*code*, int€*subcode***)**

To receive a message on an exception port, you must first create a new port and make it the task or thread exception port. (You can't use the default task exception port because you can't get receive rights for it.) Before calling **msg_receive()**, you must set the **local_port** field of the header to the appropriate exception port and the **msg_size** field to the size of the structure for the incoming message.

If it accepted the incoming message, **exc_server()** returns true; otherwise it returns false.

You should keep a global value that indicates whether your exception handler successfully handled the exception. If it couldn't, then you should forward the exception message to the old exception port.

EXAMPLE

```
typedef struct {
    port_t old_exc_port;
    port_t clear_port;
    port_t exc_port;
} ports_t;

volatile boolean_t  pass_on = FALSE;
mutex_t             printing;

/* Listen on the exception port. */
any_t exc_thread(ports_t *port_p)
{
    kern_return_t   r;
    char           *msg_data[2][64];
    msg_header_t   *imsg = (msg_header_t *)msg_data[0],
                   *omsg = (msg_header_t *)msg_data[1];

    /* Wait for exceptions. */
    while (1) {
        imsg->msg_size = 64;
        imsg->msg_local_port = port_p->exc_port;
        r = msg_receive(imsg, MSG_OPTION_NONE, 0);

        if (r==RCV_SUCCESS) {
            /* Give the message to the Mach exception server. */
            if (exc_server(imsg, omsg)) {
                /* Send the reply message that exc_serv gave us. */
                r = msg_send(omsg, MSG_OPTION_NONE, 0);
                if (r != SEND_SUCCESS) {
                    mach_error("msg_send", r);
                    exit(1);
                }
            }
            else { /* exc_server refused to handle imsg. */
                mutex_lock(printing);
                printf("exc_server didn't like the message\n");
                mutex_unlock(printing);
                exit(2);
            }
        }
        else { /* msg_receive() returned an error. */
                mach_error("msg_receive", r);
                exit(3);
        }

        /* Pass the message to old exception handler, if necessary. */
        if (pass_on == TRUE) {
            imsg->msg_remote_port = port_p->old_exc_port;
            imsg->msg_local_port = port_p->clear_port;
            r = msg_send(imsg, MSG_OPTION_NONE, 0);
            if (r != SEND_SUCCESS) {
                mach_error("msg_send to old_exc_port", r);
                exit(4);
            }
        }
    }
}

/*
 * catch_exception_raise() is called by exc_server().  The only
 * exception it can handle is EXC_SOFTWARE.
 */
kern_return_t catch_exception_raise(port_t exception_port,
    port_t thread, port_t task, int exception, int code, int subcode)
{
    if ((exception == EXC_SOFTWARE) && (code == 0x20000)) {
        /* Handle the exception so that the program can continue. */
        mutex_lock(printing);
        printf("Handling the exception\n");
        mutex_unlock(printing);
        return KERN_SUCCESS;
    }
    else { /* Pass the exception on to the old port. */
        pass_on = TRUE;
        mach_NeXT_exception("Forwarding exception", exception,
            code, subcode);
        return KERN_FAILURE;  /* Couldn't handle this exception. */
    }
}

main()
{
    int             i;
    kern_return_t   r;
    ports_t         ports;

    printing = mutex_alloc();

    /* Save the old exception port for this task. */
    r = task_get_exception_port(task_self(), &(ports.old_exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("task_get_exception_port", r);
        exit(1);
    }

    /* Create a new exception port for this task. */
    r = port_allocate(task_self(), &(ports.exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("port_allocate 0", r);
        exit(1);
    }
    r = task_set_exception_port(task_self(), (ports.exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("task_set_exception_port", r);
        exit(1);
    }

    /* Fork the thread that listens to the exception port. */
    cthread_detach(cthread_fork((cthread_fn_t)exc_thread,
        (any_t)&ports));
    /* Raise the exception. */
    ports.clear_port = thread_self();
    r = exception_raise(ports.exc_port, thread_reply(),
        ports.clear_port, task_self(), EXC_SOFTWARE, 0x20000, 6);

    if (r != KERN_SUCCESS)
        mach_error("catch_exception_raise didn't handle exception",
            r);
    else {
        mutex_lock(printing);
```

```
            printf("Successfully called exception_raise\n");
            mutex_unlock(printing);
        }
    }
```

SEE ALSO **exception_raise()**, **mach_NeXT_exception()**

exception_raise()

SUMMARY Cause an exception to occur

SYNOPSIS **#import <mach/mach.h>**
#import <mach/exception.h>

kern_return_t **exception_raise(**port_t *exception_port*, port_t *clear_port*, port_t *thread*, port_t *task*, int *exception*, int *code*, int *subcode***)**

ARGUMENTS *exception_port*: The exception port of the affected thread. (If the thread doesn't have its own exception port, then this should be the exception port of the task.)

clear_port: The port to which a reply message should be sent from the exception handler. If you don't care to see the reply, you can use **thread_reply()**.

thread: The thread in which the exception condition occurred. If the exception isn't thread-specific, then specify THREAD_NULL.

task: The task in which the exception condition occurred.

exception: The type of exception that occurred; for example, EXC_SOFTWARE. Values for this variable are defined in the header file **mach/exception.h**.

code: The exception code. The meaning of this code depends on the value of *exception*.

subcode: The exception subcode. The meaning of this subcode depends on the values of *exception* and *code*.

DESCRIPTION This function causes an exception message to be sent to *exception_port*, which results in a call to the exception handler. Usually this function is used along with a user-defined exception handler. (See **exc_server()** and **mach_NeXT_exception()** for more information on user-defined exception handlers.)

You can obtain *exception_port* by calling **thread_get_exception_port()** or (if no thread exception port exists or the exception affects the whole task) **task_get_exception_port()**.

If you're defining your own type of exception, you must have *exception* equal to EXC_SOFTWARE and *code* equal to or greater than 0x20000.

EXAMPLE

```
/* Raise the exception. */
r = exception_raise(ports.exc_port, thread_reply(), thread_self(),
    task_self(), EXC_SOFTWARE, 0x20000, 6);
if (r != KERN_SUCCESS)
    mach_error("catch_exception_raise didn't handle exception", r);
else {
    /* Use mutex so only one thread at a time can call printf. */
    mutex_lock(printing);
    printf("Successfully called exception_raise\n");
    mutex_unlock(printing);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_FAILURE: The exception handler didn't successfully deal with the exception.

KERN_INVALID_ARGUMENT: One of the arguments wasn't valid.

SEE ALSO **exc_server()**, **mach_NeXT_exception()**, **task_get_exception_port()**, **thread_get_exception_port()**

host_info()

SUMMARY Get information about a host

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **host_info(**host_t *host*, int *flavor*, host_info_t *host_info*, unsigned€int€**host_info_count***)**

ARGUMENTS *host*:   The host for which information is to be obtained.

flavor: The type of statistics to be returned.   Currently HOST_BASIC_INFO, HOST_PROCESSOR_SLOTS, and HOST_SCHED_INFO are implemented.

host_info: Returns statistics about *host*.

host_info_count: The number of integers in the info structure; returns the number of integers that Mach tried to fill the info structure with.   For HOST_BASIC_INFO, you should set *host_info_count* to HOST_BASIC_INFO_COUNT.   For HOST_PROCESSOR_SLOTS, you should set it to the maximum number of CPUs (returned by HOST_BASIC_INFO).   For HOST_SCHED_INFO, set it to HOST_SCHED_INFO_COUNT.

DESCRIPTION Returns the selected information array for a host, as specified by *flavor*.   The *host_info* argument is an array of integers that's supplied by the caller and returned filled with specified information.   The *host_info_count* argument is supplied by the caller as the maximum number of integers in *host_info* (which can be larger than the space required for the information).   On return, it contains the actual number of integers in *host_info*.

Warning: This replaces the old **host_info()** call.   It isn't backwards compatible.

Basic information is defined by HOST_BASIC_INFO.   Its size is defined by HOST_BASIC_INFO_COUNT. Possible values of the **cpu_type** and **cpu_subtype** fields are defined in the header file **mach/machine.h**, which is included in **mach/mach.h**.

```
struct host_basic_info {
    int           max_cpus;     /* maximum possible cpus for
                                 * which kernel is configured */
    int           avail_cpus;   /* number of cpus now available */
    vm_size_t     memory_size;  /* size of memory in bytes */
    cpu_type_t    cpu_type;     /* cpu type */
    cpu_subtype_t cpu_subtype;  /* cpu subtype */
};
typedef struct host_basic_info *host_basic_info_t;
```

Processor slots of the active (available) processors are defined by HOST_PROCESSOR_SLOTS.   The size of this information should be obtained from the **max_cpus** field of the structure returned by HOST_BASIC_INFO. HOST_PROCESSOR_SLOTS returns an array of integers, each of which is the slot number of a CPU.

Additional information of interest to schedulers is defined by HOST_SCHED_INFO.   The size of this information is defined by HOST_SCHED_INFO_COUNT.

```
struct host_sched_info {
    int  min_timeout;  /* minimum timeout in milliseconds */
    int  min_quantum;  /* minimum quantum in milliseconds */
};

typedef struct host_sched_info *host_sched_info_t
```

EXAMPLE An example of using HOST_BASIC_INFO:

```
kern_return_t           ret;
struct host_basic_info  basic_info;
unsigned int            count=HOST_BASIC_INFO_COUNT;

ret=host_info(host_self(), HOST_BASIC_INFO,
    (host_info_t)&basic_info, &count);
if (ret != KERN_SUCCESS)
    mach_error("host_info() call failed", ret);
else printf("This system has %d bytes of RAM.\n",
    basic_info.memory_size);
```

An example of using HOST_PROCESSOR_SLOTS (you also need to include the HOST_BASIC_INFO code above so you can get **max_cpus**):

```
host_info_t  slots;
unsigned int cpu_count, i;

cpu_count=basic_info.max_cpus;
slots=(host_info_t)malloc(cpu_count*sizeof(int));
ret=host_info(host_self(), HOST_PROCESSOR_SLOTS, slots,
    &cpu_count);
if (ret!=KERN_SUCCESS)
    mach_error("PROCESSOR host_info() call failed", ret);
else for (i=0; i<cpu_count; i++)
    printf("CPU %d is in slot %d.\n", i, *slots++);
```

An example of using HOST_SCHED_INFO:

```
kern_return_t           ret;
struct host_sched_info  sched_info;
unsigned int            sched_count=HOST_SCHED_INFO_COUNT;

ret=host_info(host_self(), HOST_SCHED_INFO,
    (host_info_t)&sched_info, &sched_count);
if (ret != KERN_SUCCESS)
    mach_error("SCHED host_info() call failed", ret);
else
    printf("The minimum quantum is %d milliseconds.\n",
        sched_info.min_quantum);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* is not a host, *flavor* is not recognized, or (for HOST_PROCESSOR_SLOTS) **count* is less than **max_cpus**.

KERN_FAILURE: **count* is less than HOST_BASIC_INFO_COUNT (when *flavor* is HOST_BASIC_INFO) or HOST_SCHED_INFO_COUNT (for HOST_SCHED_INFO).

MIG_ARRAY_TOO_LARGE: Returned info array is too large for *host_info*. The *host_info* argument is filled as much as possible, and *host_info_count* is set to the number of elements that would be returned if there were enough room.

SEE ALSO **host_kernel_version()**, **host_processors()**, **processor_info()**

host_kernel_version()

SUMMARY Get kernel version information

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **host_kernel_version(**host_t *host*, kernel_version_t *version***)**

ARGUMENTS *host*: The host for which information is being requested.

version: Returns a character string describing the kernel version executing on *host*.

DESCRIPTION This function returns the version string compiled into *host*'s kernel at the time it was built. If you don't use the **kernel_version_t** declaration, then you should allocate KERNEL_VERSION_MAX bytes for the version string.

EXAMPLE
```
kern_return_t       ret;
kernel_version_t    string;

ret=host_kernel_version(host_self(), string);
if (ret != KERN_SUCCESS)
    mach_error("host_kernel_version() call failed", ret);
else
    printf("Version string:  %s\n", string);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* was not a host.

KERN_INVALID_ADDRESS: *version* points to inaccessible memory.

SEE ALSO **host_info()**, **host_processors()**, **processor_info()**

host_processor_set_priv()

SUMMARY Get the privileged port of a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **host_processor_set_priv(**host_priv_t *host_priv*, processor_set_t€*processor_set_name*, processor_set_t **processor_set***)**

ARGUMENTS *host_priv*: The privileged host port for the desired host.

processor_set_name: The name port of the processor set.

processor_set: Returns the privileged port of the processor set.

DESCRIPTION This function returns send rights to the privileged port for the specified processor set. This port is used in calls that can affect other threads or tasks. For example, **processor_set_tasks()** requires the privileged port because it returns the port of every task on the system.

EXAMPLE
```
kern_return_t    error;
processor_set_t  processor_set;
processor_set_t  default_set;

error=processor_set_default(host_self(), &default_set);
if (error != KERN_SUCCESS)
    mach_error("Call to processor_set_default failed", error);

error=host_processor_set_priv(host_priv_self(), default_set,
    &processor_set);
if (error != KERN_SUCCESS)
    mach_error("Call to host_processor_set_priv failed; make sure
        you're superuser", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host_priv* was not a privileged host port, or *processor_set_name* didn't name a valid processor set.

host_processor_sets()

SUMMARY Get the name ports of all processor sets on a host

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **host_processor_sets(**host_t *host*, processor_set_name_array_t€**processor_set_list*, unsigned€int€**processor_set_count***)**

ARGUMENTS *host*: The host port for the desired host.

processor_set_list: Returns an array of processor sets currently existing on *host*; no particular ordering is guaranteed.

processor_set_ count: Returns the number of processor sets in the *processor_set_list*.

DESCRIPTION This function returns send rights to the name port for each processor set currently assigned to *host*. The **host_processor_set_priv()** function can be used to obtain the privileged ports from these if desired. The *processor_set_list* argument is an array that is created as a result of this call. You should call **vm_deallocate()** on this array when the data is no longer needed.

Note: In single-processor systems, you can get the same information by calling **processor_set_default()**.

EXAMPLE

```
kern_return_t               ret;
processor_set_name_array_t  list;
unsigned int                count;

ret=host_processor_sets(host_self(), &list, &count);
if (ret!=KERN_SUCCESS)
    mach_error("error calling host_processor_sets", ret);
else {
    /* . . . */
    ret=vm_deallocate(task_self(), (vm_address_t)list,
        sizeof(list)*count);
    if (ret!=KERN_SUCCESS)
        mach_error("error calling vm_deallocate", ret);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* is not a host.

SEE ALSO **host_processor_set_priv()**, **processor_set_create()**, **processor_set_tasks()**, **processor_set_threads()**, **processor_set_default()**

host_processors()

SUMMARY Get the processor ports for a host

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **host_processors(**host_priv_t *host_priv*, processor_array_t€**processor_list*, unsigned int **processor_count***)**

ARGUMENTS *host_priv*: Privileged host port for the desired host.

processor_list: Returns the processors existing on *host_priv*; no particular ordering is€guaranteed.

processor_count: Returns the number of processors in *processor_list*.

DESCRIPTION **host_processors()** gets send rights to the processor port for each processor existing on *host_priv*. The *processor_list* argument is an array that is created as a result of this call. The caller may wish to call **vm_deallocate()** on this array when the data is no longer needed.

EXAMPLE
```
kern_return_t      error;
processor_array_t  list;
unsigned int       count;

error=host_processors(host_priv_self(), &list, &count);
if (error!=KERN_SUCCESS){
    mach_error("error calling host_processors", error);
    exit(1);
}
/* . . . */
vm_deallocate(task_self(), (vm_address_t)list, sizeof(list)*count);
if (error!=KERN_SUCCESS)
    mach_error("Trouble freeing list", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host_priv* is not a privileged host port.

SEE ALSO **processor_info()**, **processor_start()**, **processor_exit()**, **processor_control()**

host_self(), host_priv_self()

SUMMARY Get the host port for this host

SYNOPSIS **#import <mach/mach.h>**

host_t **host_self(**void**)**
host_priv_t **host_priv_self(**void**)**

DESCRIPTION The **host_self()** function returns send rights to the host port for the host on which the call is executed. This port can be used only to obtain information about the host, not to control the host.

The **host_priv_self()** function returns send rights to the privileged host port for the host on which the call is executed. This port is used to control physical resources on that host and is only available to privileged tasks. PORT_NULL is returned if the invoker is not the UNIX superuser.

EXAMPLE
```
/* Get the privileged port for the default processor set. */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error!=KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}
```

SEE ALSO **host_processors()**, **host_info()**, **host_kernel_version()**

mach_error(), mach_error_string()

SUMMARY Display or get a Mach error string

SYNOPSIS **#import <mach/mach.h>**
#import <mach/mach_error.h>

void **mach_error(**char **string*, kern_return_t *error***)**
char ***mach_error_string(**kern_return_t *error***)**

ARGUMENTS *string*: The string you want displayed before the Mach error string.

error: The error value for which you want an error string.

DESCRIPTION The function **mach_error()** displays a message on **stderr**. The message contains the string specified by *string*, the string returned by **mach_error_string()**, and the actual error value (*error*). Since **mach_error()** isn't thread-safe, you might want to protect it with a mutex if you call it in a multiple-thread task.

The function **mach_error_string()** returns the string associated with *error*.

Note that because the error value specified by *error* is of type **kern_return_t**, these functions work only with Mach functions.

EXAMPLE

```
mutex_t        printing;

main()
{
    kern_return_t  error;
    port_t         result;

    printing = mutex_alloc();

    /* . . . */
    if ((error=port_allocate(task_self(), &result)) != KERN_SUCCESS) {
        mutex_lock(printing);
        mach_error("Error calling port_allocate", error);
        mutex_unlock(printing);
        exit(1);
    }
    /* . . . */
}
```

mach_NeXT_exception(), mach_NeXT_exception_string()

SUMMARY Display or get a Mach exception string

SYNOPSIS **#import <mach/mach.h>**

void **mach_NeXT_exception(**char **string*, int *exception*, int *code*, int *subcode***)**
char ***mach_NeXT_exception_string(**int *exception*, int *code*, int *subcode***)**

ARGUMENTS *string*: The string you want displayed before the Mach exception string.

exception: The exception value for which you want a string.

code: The exception code. How this is used depends on the value of *exception*.

subcode: The exception subcode. How this is used depends on the value of *exception*.

DESCRIPTION The function **mach_NeXT_exception()** displays a message on **stderr**. The message contains the string specified by *string*, then the string returned by **mach_NeXT_exception_string()**, and then the values of *exception*, *code*, and *subcode*. Since **mach_NeXT_exception()** isn't thread-safe, you might want to protect it with a mutex if you call it in a multiple-thread task.

The function **mach_NeXT_exception_string()** returns the string associated with *exception*, *code*, and *subcode*.

EXAMPLE

```
/*
 * catch_exception_raise() is called by exc_server().  The only
 * exception it can handle is EXC_SOFTWARE.
 */
kern_return_t catch_exception_raise(port_t exception_port,
    port_t thread, port_t task, int exception, int code, int subcode)
{
    if ((exception == EXC_SOFTWARE) && (code == 0x20000)) {
        /* Handle the exception so that the program can continue. */
        mutex_lock(printing);
        printf("Handling the exception\n");
        mutex_unlock(printing);
        return KERN_SUCCESS;
    }
    else { /* Pass the exception on to the old port. */
        pass_on = TRUE;
        mach_NeXT_exception("Forwarding exception", exception,
            code, subcode);
        return KERN_FAILURE;  /* Couldn't handle this exception. */
    }
}
```

SEE ALSO **exception_raise()**, **exc_server()**

map_fd()

SUMMARY Map a file into virtual memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **map_fd(**int *fd*, vm_offset_t *offset*, vm_offset_t **address*, boolean_t€*find_space*, vm_size_t *size***)**

ARGUMENTS *fd*: An open UNIX file descriptor for the file that's to be mapped.

offset: The byte offset within the file, at which mapping is to begin.

address: A pointer to an address in the calling process at which the mapped file should start. This address, unlike the offset, must be page-aligned.

find_space: If true, the kernel will select an unused address range at which to map the file and return its value in *address*.

size: The number of bytes to be mapped.

DESCRIPTION The function **map_fd()** is a UNIX extension that's technically not part of Mach. This function causes *size* bytes of data starting at *offset* in the file specified by *fd* to be mapped into the virtual memory at the address specified by *address*. If *find_space* is true, the input value of *address* can be null, and the kernel will find an unused piece of virtual memory to use. (You should free this space with **vm_deallocate()** when you no longer need it.) If you provide a value for *address*, it must be page-aligned and at least *size* bytes long. The sum of *offset* and *size* must not exceed the length of the file.

Memory mapping doesn't cause I/O to take place. When specific pages are first referenced, they cause page faults that bring in the data. The mapped memory is copy-on-write. Modified data is returned to the file only by a **write()** call.

EXAMPLE

```
kern_return_t r;
```

```
int            fd;
char          *memfile, *filename = "/tmp/myfile";

/* Open the file. */
fd = open(filename, O_RDONLY);

/* Map part of it into memory. */
r = map_fd(fd, (vm_offset_t)0, &(vm_offset_t)memfile, TRUE,
    (vm_size_t)5);
if (r != KERN_SUCCESS)
    mach_error("Error calling map_fd()", r);
else
    printf("Second character in %s is:  %c\n", filename, memfile[1]);
```

RETURN KERN_SUCCESS: The data was mapped successfully.

KERN_INVALID_ADDRESS: *address* wasn't valid.

KERN_INVALID_ARGUMENT: An invalid argument was passed.

msg_receive()

SUMMARY Receive a message

SYNOPSIS **#import <mach/mach.h>**
#import <mach/message.h>

msg_return_t **msg_receive(**msg_header_t **header*, msg_option_t *option*, msg_timeout_t€*timeout***)**

ARGUMENTS *header*: The address of a buffer in which the message is to be received. Two fields of the message header must be set before the call is made: **msg_local_port** must be set to the value of the port from which the message is to be received, and **msg_size** must be set to the maximum size of the message that may be received. This maximum size must be less than or equal to the size of the buffer.

option: The failure conditions under which **msg_receive()** should terminate. The value of this argument is a combination (using the bitwise OR operator) of the following options. Unless one of these values is explicitly specified, **msg_receive()** does not return until a message has been received.

RCV_TIMEOUT: Specifies that **msg_receive()** should return when the specified timeout elapses if a message has not arrived by that time; if not specified, the timeout will be ignored (that is, it will be infinite).

RCV_INTERRUPT: Specifies that **msg_receive()** should return when a software interrupt occurs in this thread.

RCV_LARGE: Specifies that **msg_receive()** should return without dequeuing a message if the next message in the queue is larger than *header***.msg_size**. (Normally, a message that is too large is dequeued and lost.) You can use this option to dynamically determine how large your message buffer must be.

Use MSG_OPTION_NONE to specify that none of the above options is desired.

timeout: If RCV_TIMEOUT is specified in *option*, then *timeout* is the maximum time in milliseconds to wait for a message before giving up.

DESCRIPTION The function **msg_receive()** retrieves the next message from the port or port set specified in the **msg_local_port** field of *header*. If a port is specified, the port must not be a member of a port set.

If a port set is specified, then **msg_receive()** will retrieve messages sent to any of the set's€member ports. Mach sets the **msg_local_port** field to the specific port on which the€message was found. It's not an error for the port set to have no members, or for members to be added and removed from a port set while a **msg_receive()** on the port set is€in progress.

The message consists of its header, followed by a variable amount of data; the message header supplied to

msg_receive() must specify (in **msg_size**) the maximum size of the message that can be received into the buffer provided.

If no messages are present on the port(s) in question, **msg_receive()** will wait until a message arrives, or until one of the specified termination conditions is met (see the description of the *option* argument for this function).

If the message is successfully received, then **msg_receive()** sets the **msg_size** field of the header to the size of the received message. If the RCV_LARGE option was set and **msg_receive()** returned RCV_TOO_LARGE, then the **msg_size** field is set to the size of the message that was too large.

If the received message contains out-of-line data (that is, data for which the **msg_type_inline** attribute was specified as false), the data will be returned in a newly allocated region of memory; the message body will contain a pointer to that new region. You should deallocate this memory when the data is no longer needed. See the **vm_allocate()** call for a description of the state of newly allocated memory.

See Chapter 2, ªUsing Mach Messages,º for information on setting up messages and on writing Mach servers.

EXAMPLE

```
msg_header_t   *imsg, header;

/* Wait for messages. */
while (1) {
    /* Set up the message structure. */
    header.msg_size = sizeof header;
    header.msg_local_port = receive_port;

    /* Get the next message on the queue. */
    r = msg_receive(&header, RCV_LARGE, 0);

    /* If the message is too big ... */
    if (r==RCV_TOO_LARGE) {
        /* ... allocate a structure for it ... */
        imsg = (msg_header_t *)malloc(header.msg_size);
        /* ... initialize the structure ... */
        imsg->msg_size = header.msg_size;
        imsg->msg_local_port = receive_port;
        /* ... and get the message. */
        r = msg_receive(imsg, MSG_OPTION_NONE, 0);
    }

    if (r==RCV_SUCCESS) {
        /* Handle the message. */
    }
    else { /* msg_receive() returned an error. */
        mach_error("msg_receive", r);
        exit(3);
    }
}
```

RETURN RCV_SUCCESS: The message has been received.

RCV_INVALID_MEMORY: The message specified was not writable by the calling task.

RCV_INVALID_PORT: An attempt was made to receive on a port to which the calling task does not have the proper access, or which was deallocated (see **port_deallocate()**) while waiting for a message.

RCV_TOO_LARGE: The message header and body combined are larger than the size specified by **msg_size**. Unless the RCV_LARGE option was set, the message has been dequeued and lost. If the RCV_LARGE option was specified, then Mach sets **msg_size** to the size of the message that was too large and leaves the message at the head of the queue.

RCV_NOT_ENOUGH_MEMORY: The message to be received contains more out-of-line data than can be allocated in the receiving task.

RCV_TIMED_OUT: The message was not received after *timeout* milliseconds.

RCV_INTERRUPTED: A software interrupt occurred and the RCV_INTERRUPT option was specified.

RCV_PORT_CHANGE: The port specified was added to a port set during the duration of the **msg_receive()** call.

msg_rpc()

SUMMARY Send and receive a message

SYNOPSIS **#import <mach/mach.h>**
#import <mach/message.h>

msg_return_t **msg_rpc(**msg_header_t **header*, msg_option_t *option*, msg_size_t€*rcv_size*, msg_timeout_t *send_timeout*, msg_timeout_t *rcv_timeout***)**

ARGUMENTS *header*: Address of a message buffer that will be used for both **msg_send()** and **msg_receive()**. This buffer contains a message header followed by the data for the message to be sent. The **msg_remote_port** field specifies the port to which the message is to be sent. The **msg_local_port** field specifies the port on which a message is then to be received; if this port is the special value PORT_DEFAULT, it gets replaced by the value PORT_NULL for the purposes of the **msg_send()** operation.

option: A union of the *option* arguments for the send and receive (see **msg_send()** and **msg_receive()**).

rcv_size: The maximum size allowed for the received message; this must be less than or equal to the size of the message buffer. The **msg_size** field in the header specifies the size of the message to be sent.

send_timeout, *rcv_timeout*: The timeout values to be applied to the component operations. These are used only if the option SEND_TIMEOUT or RCV_TIMEOUT is specified.

DESCRIPTION The function **msg_rpc()** is a hybrid call that performs a **msg_send()** followed by a **msg_receive()**, using the same message buffer. Because of the order of the send and receive, this function is appropriate for clients of Mach servers. However, the **msg_rpc()** call to a Mach server is usually performed by MiG-generated code, not by handwritten code.

See Chapter 2, ªUsing Mach Messages,º for information on setting up messages and on writing Mach servers.

RETURN RPC_SUCCESS: The message was successfully sent and a reply was received.

Other possible values are the same as those for **msg_send()** and **msg_receive()**; any error during the **msg_send()** portion will terminate the call.

msg_send()

SUMMARY Send a message

SYNOPSIS **#import <mach/mach.h>**
#import <mach/message.h>

msg_return_t **msg_send(**msg_header_t **header*, msg_option_t *option*, msg_timeout_t€*timeout***)**

ARGUMENTS *header*: The address of the message to be sent. A message consists of a fixed-size header followed by a variable number of data descriptors and data items. See the header file **mach/message.h** for a definition of the message structure.

option: The failure conditions under which **msg_send()** should terminate. The value of this argument is a combination (using the bitwise OR operator) of the following options. Unless one of these values is explicitly specified, **msg_send()** does not return until the message is successfully queued for the intended receiver.

SEND_TIMEOUT: Specifies that the **msg_send()** request should terminate after the timeout period has elapsed, even if the kernel has been unable to queue the message.

SEND_NOTIFY: Allows the sender to send exactly one message without being suspended even if the destination port is full. When that message can be posted to the receiving port queue, this task receives a message that notifies it that another message can be sent. If the sender tries to send a second message with this option to the same port before the first notification arrives, the result is an error. If both SEND_NOTIFY and SEND_TIMEOUT are specified, **msg_send()** will wait until the specified timeout has elapsed before invoking the SEND_NOTIFY option.

SEND_INTERRUPT: Specifies that **msg_send()** should return if a software interrupt occurs in this thread.

Use MSG_OPTION_NONE to specify that none of the above options is wanted.

timeout: If the destination port is full and the SEND_TIMEOUT option has been specified, this value specifies the maximum wait time (in milliseconds).

DESCRIPTION The function **msg_send()** transmits a message from the current task to the port specified in the message header field. The message consists of its header, followed by a variable number of data descriptors and data items.

If the **msg_local_port** field isn't set to PORT_NULL, send rights to that port will be passed to the receiver of this message. The receiver task can use that port to send a reply to this message.

If the SEND_NOTIFY option is used and this call returns a SEND_WILL_NOTIFY code, you can expect to receive a notify message from the kernel. This message will be either a NOTIFY_MSG_ACCEPTED or a NOTIFY_PORT_DELETED message, depending on what happened to the queued message. The **notify_port** field in these messages is the port to which the original message was sent. The formats for these messages are defined in the header file **sys/notify.h**.

See Chapter 2, ªUsing Mach Messages,º for information on setting up messages and on writing Mach servers.

EXAMPLE

```
/* From the handwritten part of a Mach server... */
while (TRUE)
{
    /* Receive a request from a client. */
    msg.head.msg_local_port = port;
    msg.head.msg_size = sizeof(struct message);
    ret = msg_receive(&msg.head, MSG_OPTION_NONE, 0);
    if (ret != RCV_SUCCESS) /* ignore errors */;

    /* Feed the request into the server. */
    (void)add_server(&msg, &reply);

    /* Send a reply to the client. */
    reply.head.msg_local_port = port;
    ret = msg_send(&reply.head, MSG_OPTION_NONE, 0);
    if (ret != SEND_SUCCESS) /* ignore errors */;
}
```

RETURN SEND_SUCCESS: The message has been queued for the destination port.

SEND_INVALID_MEMORY: The message header or body was not readable by the calling task, or the message body specified out-of-line data that was not readable.

SEND_INVALID_PORT: The message refers either to a port for which the current task does not have access, or to which access was explicitly removed from the current task (see **port_deallocate()**) while waiting for the message to be posted, or a **msg_type_name** field in the message specifies rights that the name doesn't denote in the task (for example, specifying MSG_TYPE_SEND and supplying a port set name).

SEND_TIMED_OUT: The message was not sent since the destination port was still full after *timeout* milliseconds.

SEND_WILL_NOTIFY: The destination port was full but the SEND_NOTIFY option was specified. A notification message will be sent when the message can be posted.

SEND_NOTIFY_IN_PROGRESS: The SEND_NOTIFY option was specified but a notification request is already outstanding for this thread and given destination port.

port_allocate()

SUMMARY Create a port

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_allocate(**task_t *task*, port_name_t **port_name***)**

ARGUMENTS *task*: The task in which the new port is created (for example, use **task_self()** to specify the caller's task).

port_name: Returns the name used by *task* for the new port.

DESCRIPTION The function **port_allocate()** causes a port to be created for the specified task; the resulting port is returned in *port_name.* The target task initially has both send and receive rights to the port. The new port isn't a member of any port set.

EXAMPLE

```
port_t          myport;
kern_return_t  error;

if ((error=port_allocate(task_self(), &myport)) != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: A port has been allocated.

KERN_INVALID_ARGUMENT: *task* was invalid.

KERN_RESOURCE_SHORTAGE: No more port slots are available for this task.

SEE ALSO **port_deallocate()**

port_deallocate()

SUMMARY Deallocate a port

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_deallocate(**task_t *task*, port_name_t *port_name***)**

ARGUMENTS *task*: The task that wants to relinquish rights to the port (for example, use **task_self()** to specify the caller's task).

port_name: The name that *task* uses for the port to be deallocated.

DESCRIPTION The function **port_deallocate()** requests that the target task's access to a port be relinquished.

If *task* has receive rights for the port and the port doesn't have a backup port, these things happen:

- The port is destroyed.
- All other tasks with send access to the port are notified of its destruction.
- If the port is a member of a port set, it's removed from the port set.

If *task* has receive rights for the port and the port *does* have a backup port, then the following things happen:

- If the port is a member of a port set, it's removed from the port set.
- Send and receive rights for the port are sent to the backup port in a notification message (see **port_set_backup()**).

EXAMPLE
```
port_t          my_port;
kern_return_t  error;

/* . . . */

error=port_deallocate(task_self(), my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_deallocate failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The port has been deallocated.

KERN_INVALID_ARGUMENT: *task* was invalid or *port_name* doesn't name a valid€port.

SEE ALSO **port_allocate()**

port_extract_receive(), port_extract_send()

SUMMARY Remove access rights to a port and return them to the caller

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_extract_receive(**task_t *task*, port_name_t *its_name*, port_t€**its_port***)**
kern_return_t **port_extract_send(**task_t *task*, port_name_t *its_name*, port_t€**its_port***)**

ARGUMENTS *task*: The task whose rights the caller takes.

its_name: The name by which *task* knows the port.

its_port: Returns the receive or send rights.

DESCRIPTION The functions **port_extract_receive()** and **port_extract_send()** remove the port access rights that *task* has for a port and return the rights to the caller. This leaves *task* with no rights for the port.

The **port_extract_send()** function extracts send rights; *task* can't have receive rights for the named port. The **port_extract_receive()** function extracts receive rights.

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *its_name* doesn't name a port for which *task* has the required rights.

SEE ALSO **port_insert_send()**, **port_insert_receive()**

port_insert_receive(), port_insert_send()

SUMMARY Give a task rights with a specific name

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_insert_receive(**task_t *task*, port_t *my_port*, port_name_t€*its_name***)** kern_return_t **port_insert_send(**task_t *task*, port_t *my_port*, port_name_t *its_name***)**

ARGUMENTS *task*: The task getting the new rights.

my_port: Rights supplied by the caller.

its_name: The name by which *task* will know the new rights.

DESCRIPTION The functions **port_insert_receive()** and **port_insert_send()** give a task rights with a specific name. If *task* already has rights named *its_name*, or has some other name for *my_port*, the operation will fail. The *its_name* argument can't be a predefined port, such as€PORT_NULL.

The **port_insert_send()** function inserts send rights, and **port_insert_receive()** inserts receive rights.

RETURN KERN_SUCCESS: The call succeeded.

KERN_NAME_EXISTS: *task* already has a right named *its_name*.

KERN_FAILURE: *task* already has rights to *my_port*.

KERN_INVALID_ARGUMENT: *task* was invalid or *its_name* was an invalid name.

SEE ALSO **port_extract_send()**, **port_extract_receive()**

port_names()

SUMMARY Get information about the port name space of a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_names(**task_t *task*, port_name_array_t **port_names*, unsigned€int€**port_names_count*, port_type_array_t **port_types*, unsigned€int€**port_types_count***)**

ARGUMENTS *task*: The task whose port name space is queried.

port_names: Returns the names of the ports and port sets in the port name space of *task*, in no particular order.

port_names_count: Returns the number of names returned.

port_types: Returns the type of each corresponding name. This indicates what kind of rights the task holds for the port, or whether the name refers to a port set. The type is one of the following: PORT_TYPE_SEND (send rights only), PORT_TYPE_RECEIVE_OWN (send and receive rights), PORT_TYPE_SET (the port is a port set).

port_types_count: Returns the same value as *port_names_count*.

DESCRIPTION The function **port_names()** returns information about the port name space of *task*. It returns the port and port set names that are currently valid for *task*. For each name, it also returns what type of rights *task* holds.

The *port_names* and *port_types* arguments are arrays that are automatically allocated when the reply message is received. You should use **vm_deallocate()** on them when the data is no longer needed.

EXAMPLE
```
kern_return_t      error;
port_name_array_t  names;
unsigned int       names_count, types_count;
port_type_array_t  types;

error=port_names(task_self(), &names, &names_count, &types,
    &types_count);
if (error != KERN_SUCCESS) {
    mach_error("port_rename returned value of ", error);
```

```
        exit(1);
    }
    /* . . . */
    error=vm_deallocate(task_self(), (vm_address_t)names,
        sizeof(names)*names_count);
    if (error != KERN_SUCCESS)
        mach_error("Trouble freeing names", error);

    error=vm_deallocate(task_self(), (vm_address_t)types,
        sizeof(names)*types_count);
    if (error != KERN_SUCCESS)
        mach_error("Trouble freeing types", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid.

SEE ALSO **port_type(), port_status(), port_set_status()**

port_rename()

SUMMARY Change the name by which a port or port set is known to a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_rename(**task_t *task*, port_name_t *old_name*, port_name_t€*new_name***)**

ARGUMENTS *task*: The task whose port name space is changed.

old_name: The current name of the port or port set.

new_name: The new name for the port or port set.

DESCRIPTION The function **port_rename()** changes the name by which a port or port set is known to *task.* The port name specified in *new_name* must not already be in use, and it can't be a predefined port, such as PORT_NULL. Currently, a name is a small integer.

One way to guarantee that a name isn't already in use is to deallocate a port and then use its name as *new_name*. Another way is to check all the existing names, using **port_names()**, before you call **port_rename()**. If you choose another naming scheme, you should be prepared to try another name if **port_rename()** returns a KERN_NAME_EXISTS error.

EXAMPLE

```
#define MY_PORT  (port_name_t)99

port_name_t       my_port;
kern_return_t     error;

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_rename(task_self(), my_port, MY_PORT);
if (error == KERN_NAME_EXISTS)
    /* try again with a different name */;
else if (error != KERN_SUCCESS) {
        mach_error("port_rename failed", error);
        exit(1);
    }
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_NAME_EXISTS: *task* already has a port or port set named *new_name.*

KERN_INVALID_ARGUMENT: *task* was invalid, or *task* didn't know any ports or port sets named *old_name*, or *new_name* was an invalid name.

SEE ALSO **port_names()**

port_set_add()

SUMMARY Move the named port into the named port set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_add(**task_t *task*, port_set_name_t *set_name*, port_name_t€*port_name***)**

ARGUMENTS *task*: The task that has receive rights for the port set and port.

set_name: *task*'s name for the port set.

port_name: *task*'s name for the port.

DESCRIPTION The function **port_set_add()** moves the named port into the named port set. The *task* must have receive rights for the port. If the port is already a member of another port set, it's removed from that set first.

EXAMPLE

```
kern_return_t    error;
port_set_name_t  set_name;
port_t           my_port;

error=port_set_allocate(task_self(),&set_name);
if (error != KERN_SUCCESS) {
    mach_error("port_set_allocate failed", error);
    exit(1);
}

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_set_add(task_self(), set_name, my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for the port.

KERN_INVALID_ARGUMENT: *task* was invalid, or *set_name* doesn't name a valid port set, or *port_name* doesn't name a valid port.

SEE ALSO **port_set_remove()**

port_set_allocate()

SUMMARY Create a port set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_allocate(**task_t *task*, port_set_name_t **set_name***)**

ARGUMENTS *task*: The task in which the new port set is created.

set_name: Returns *task*'s name for the new port set.

DESCRIPTION The function **port_set_allocate()** causes a port set to be created for the specified task; the resulting set's name is returned in *set_name.* The new port set is empty.

EXAMPLE

```
kern_return_t    error;
port_set_name_t  set_name;

error=port_set_allocate(task_self(),&set_name);
if (error != KERN_SUCCESS) {
    mach_error("port_set_allocate failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid.

KERN_RESOURCE_SHORTAGE: The kernel ran out of memory.

SEE ALSO **port_set_deallocate(), port_set_add()**

port_set_backlog()

SUMMARY Set the size of the port queue

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_backlog(**task_t *task*, port_name_t *port_name*, int *backlog***)**

ARGUMENTS *task*: The task that has receive rights for the named port (for example, use **task_self()** to specify the caller's task).

port_name: *task*'s name for the port.

backlog: The new backlog to be set.

DESCRIPTION The function **port_set_backlog()** changes the backlog value on the specified port (the port's backlog value is the number of unreceived messages that are allowed in its message queue before the kernel will refuse to accept any more sends to that port).

The task specified by *task* must have receive rights for the named port.

The maximum backlog value is the constant PORT_BACKLOG_MAX. You can get a port's current backlog value by calling **port_status()**.

EXAMPLE

```
#define MY_BACKLOG  10

kern_return_t   error;
port_t          my_port;

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
```

```
        exit(1);
    }

    error=port_set_backlog(task_self(), my_port, MY_BACKLOG);
    if (error!=KERN_SUCCESS)
        mach_error("Call to port_set_backlog failed", error);
```

RETURN KERN_SUCCESS: The backlog value has been changed.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for the port.

KERN_INVALID_ARGUMENT: *task* was invalid, or *port_name* doesn't name a valid port, or the desired backlog wasn't greater than 0, or the desired backlog was greater than PORT_BACKLOG_MAX.

SEE ALSO **msg_send(), port_status()**

port_set_backup()

SUMMARY Set the backup port for a port

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_backup(**task_t *task*, port_name_t *port_name*, port_t *backup*, port_t **previous***)**

ARGUMENTS *task*: The task that has receive rights for the named port (for example, use **task_self()** to specify the caller's task).

port_name: *task*'s name for the port right.

backup: The new backup port. If you want to disable the current backup port without setting a new one, set this to PORT_NULL.

previous: Returns the previous backup port.

DESCRIPTION Use this function to keep a port alive despite its being deallocated by its receiver. If the call to **port_set_backup()** is successful, then whenever *port_name* is deallocated by its receiver, *backup* will receive a notification message with receive and send rights for *port_name*. As far as *task* is concerned, the port will be deleted; however, as far as senders to the port are concerned, the port will continue to exist.

To let a port die naturally after its backup port has been set, call **port_set_backup()** on it with *backup* set to PORT_NULL.

EXAMPLE

```
kern_return_t    error;
port_t           my_port, backup_port, previous_port;

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_allocate(task_self(),&backup_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_set_backup(task_self(), my_port, backup_port,
    &previous_port);
if (error!=KERN_SUCCESS)
    mach_error("Call to port_set_backlog failed", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid, or *port_name* doesn't name a valid€port.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for *port_name*.

port_set_deallocate()

SUMMARY Destroy a port set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_deallocate(**task_t *task*, port_set_name_t *set_name***)**

ARGUMENTS *task*: The task that has receive rights for the port set to be destroyed.

set_name: *task*'s name for the doomed port set.

DESCRIPTION The function **port_set_deallocate()** requests that the port set of *task* be destroyed. If the port set isn't empty, any members are first removed.

EXAMPLE
```
kern_return_t   error;
port_set_name_t set_name;

error=port_set_deallocate(task_self(),set_name);
if (error != KERN_SUCCESS) {
    mach_error("port_set_deallocate failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *set_name* doesn't name a valid port set.

SEE ALSO **port_set_allocate()**

port_set_remove()

SUMMARY Remove the named port from a port set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_remove(**task_t *task*, port_name_t *port_name***)**

ARGUMENTS *task*: The task that has receive rights for the port and port set.

port_name: *task*'s name for the receive rights to be removed.

DESCRIPTION The function **port_set_remove()** removes the named port from a port set. The *task* must have receive rights for the port, and the port must be a member of a port set.

EXAMPLE
```
error=port_set_remove(task_self(), my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_set_remove failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for the port.

KERN_NOT_IN_SET: The port isn't a member of a set.

KERN_INVALID_ARGUMENT: *task* was invalid or *port_name* doesn't name a valid€port.

SEE ALSO **port_set_add()**

port_set_status()

SUMMARY Get the members of a port set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_status(**task_t *task*, port_set_name_t *set_name*, port_name_array_t€**members*, unsigned int **members_count***)**

ARGUMENTS *task*: The task whose port set is queried.

set_name: *task's* name for the port set.

members: Returns *task*'s names for the members of its port set.

members_count: Returns the number of port names in *members*.

DESCRIPTION The function **port_set_status()** returns a list of the ports in a port set. The *members* argument is an array that's automatically allocated when the reply message is received. You€should use **vm_deallocate()** on it when the data is no longer needed.

EXAMPLE
```
error=port_set_status(task_self(), set_name, &members,
    &members_count);
if (error != KERN_SUCCESS) {
    mach_error("port_set_status failed", error);
    exit(1);
}

/* . . . */
error=vm_deallocate(task_self(), (vm_address_t)members,
    sizeof(members)*members_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing members", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *set_name* doesn't name a valid port€set.

SEE ALSO **port_status()**

port_status()

SUMMARY Examine a port's current status

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_status(**task_t *task*, port_name_t *port_name*, port_set_name_t€**port_set_name*, int **num_msgs*, int **backlog*, boolean_t€**owner*, boolean_t€**receiver***)**

ARGUMENTS *task*: The task that has receive rights for the port in question (for example, use **task_self()** to specify the caller's task).

port_name: *task*'s name for the port right.

port_set_name: Returns *task*'s name for the port set that the named port belongs to, or PORT_NULL if it isn't in a set.

num_msgs: Returns the number of messages queued on this port. If *task* isn't the port's receiver, the number of messages will be returned as negative.

backlog: Returns the number of messages that can be queued to this port without causing the sender to block.

owner: Returns the same value as *receiver*, since ownership rights and receive rights aren't€separable.

receiver: Returns true if *task* has receive rights to *port_name*; otherwise, returns false.

DESCRIPTION The function **port_status()** returns the current port status associated with *port_name*.

EXAMPLE
```
error=port_status(task_self(), my_port, &port_set_name, &num_msgs,
    &backlog, &owner, &receiver);
if (error!=KERN_SUCCESS)
    mach_error("Call to port_status failed", error);
```

RETURN KERN_SUCCESS: The data has been retrieved.

KERN_INVALID_ARGUMENT: *task* was invalid or *port_name* doesn't name a valid€port.

SEE ALSO **port_set_backlog()**, **port_set_status()**

port_type()

SUMMARY Determine the access rights of a task for a specific port name

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_type(**task_t *task*, port_name_t *port_name*, port_type_t **port_type***)**

ARGUMENTS *task*: The task whose port name space is queried.

port_name: The name being queried.

port_type: Returns a value that indicates what kind of rights the task holds for the port, or€whether the name refers to a port set. This value is one of the following: PORT_TYPE_SEND (send rights only), PORT_TYPE_RECEIVE_OWN (send and receive rights), PORT_TYPE_SET (the port is a port set).

DESCRIPTION The function **port_type()** returns information about *task*'s rights for a specific name in its port name space.

EXAMPLE
```
error=port_type(task_self(), port, &type);
if (error != KERN_SUCCESS)
    mach_error("Couldn't get type of port", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *task* didn't have any rights named *port_name*.

SEE ALSO **port_names()**, **port_status()**, **port_set_status()**

processor_assign(), processor_control(), processor_exit(), processor_get_assignment(), processor_start()

SUMMARY Control a processor

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_assign(**processor_t *processor*, processor_set_t€*new_processor_set*, boolean_t *wait***)**
kern_return_t **processor_control(**processor_t *processor*, processor_info_t *info*, long€**count***)**
kern_return_t **processor_exit(**processor_t *processor***)**
kern_return_t **processor_get_assignment(**processor_t *processor*, processor_set_t€**processor_set***)**
kern_return_t **processor_start(**processor_t *processor***)**

DESCRIPTION **processor_assign()** changes the processor set to which *processor* is assigned. **processor_control()** returns information about *processor*. **processor_exit()** shuts down€*processor*. **processor_get_assignment()** returns the processor set to which *processor* is assigned. **processor_start()** starts up *processor*.

Note: These functions are useful only on multiprocessor systems.

processor_info()

SUMMARY Get information about a processor

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_info(**processor_t *processor*, int *flavor*, host_t **host*, processor_info_t *processor_info*, unsigned int **processor_info_count***)**

ARGUMENTS *processor*: The processor for which information is to be obtained.

flavor: The type of information that is wanted. Currently only PROCESSOR_BASIC_INFO is implemented.

host: Returns the non-privileged host port for the host on which the processor resides.

processor_info: Returns information about the processor specified by *processor*.

processor_info_count: Size of the info structure. Should be PROCESSOR_BASIC_INFO_COUNT when *flavor* is PROCESSOR_BASIC_INFO.

DESCRIPTION Returns the selected information array for a processor, as specified by *flavor*. The *processor_info* argument is an array of integers that is supplied by the caller and filled with specified information. The *processor_info_count* argument is supplied as the maximum number of integers in *processor_info*. On return, it contains the actual number of integers in *processor_info*.

Basic information is defined by PROCESSOR_BASIC_INFO. The size of this information is defined by PROCESSOR_BASIC_INFO_COUNT. The data structures used by PROCESSOR_BASIC_INFO are defined in the header file **mach/processor_info.h**. Possible values of the **cpu_type** and **cpu_subtype** fields are defined in the header file **mach/machine.h**.

```
typedef int *processor_info_t;  /* variable length array of int */

/* one interpretation of info is */
struct processor_basic_info {
    cpu_type_t     cpu_type;    /* cpu type */
    cpu_subtype_t  cpu_subtype; /* cpu subtype */
    boolean_t      running;     /* is processor running? */
    int            slot_num;    /* slot number */
    boolean_t      is_master;   /* is this the master processor */
};

typedef struct processor_basic_info *processor_basic_info_t;
```

EXAMPLE

```
kern_return_t                   error;
host_t                      host;
unsigned int                list_size, info_count;
struct processor_basic_info info;
processor_array_t           list;

/* Get the processor port. */
error=host_processors(host_priv_self(), &list, &list_size);
if ((error!=KERN_SUCCESS) || (list_size < 1)){
    mach_error("Error calling host_processors (are you root?)",
        error);
    exit(1);
}

/* Get information about the processor. */
info_count=PROCESSOR_BASIC_INFO_COUNT;
error=processor_info(list[0], PROCESSOR_BASIC_INFO, &host,
    (processor_info_t)&info, &info_count);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_info", error);

/* Now that we're done with the processor port, free it. */
vm_deallocate(task_self(), (vm_address_t)list,
    sizeof(list)*list_size);
if (error!=KERN_SUCCESS)
    mach_error("Trouble freeing list", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor* isn't a known processor.

MIG_ARRAY_TOO_LARGE: Returned info array is too large for *processor_info*. The *processor_info* argument is filled as much as possible, and *processor_info_count* is set to the number of elements that would be returned if there were enough room.

KERN_FAILURE: *flavor* isn't recognized or *processor_info_count* is too small.

SEE ALSO **processor_start()**, **processor_exit()**, **processor_control()**, **host_processors()**

processor_set_create()

SUMMARY Create a new processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_create(**host_t *host*, port_t **new_set*, port_t **new_name***)**

DESCRIPTION This function creates a new processor set on *host*.

Note: This function is useful only on multiprocessor systems.

processor_set_default()

SUMMARY Get the port of the default processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_default(**host_t *host*, processor_set_t **default_set*)

ARGUMENTS *host*: The host whose default processor set is requested.

default_set: Returns the name (nonprivileged) port for the default processor set.

DESCRIPTION The default processor set is used by all threads, tasks, and processors that aren't explicitly assigned to other sets. This function returns a port that can be used to obtain information about this set (for example, how many threads are assigned to it). This port isn't privileged and thus can't be used to perform operations on that set; call **host_processor_set_priv()** after **processor_set_default()** to get the privileged port.

EXAMPLE
```
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS){
    mach_error("Error calling processor_set_default", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* is not a host.

SEE ALSO **processor_set_info()**, **task_assign()**, **thread_assign()**

processor_set_destroy()

SUMMARY Delete a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_destroy(**processor_set_t *processor_set***)**

DESCRIPTION This function destroys *processor_set*, reassigning all of its tasks, threads, and processors to the default processor set.

Note: This function is useful only on multiprocessor systems.

processor_set_info()

SUMMARY Get information about a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_info(**processor_set_t *processor_set*, int *flavor*, host_t€**host*, processor_set_info_t *processor_set_info*, unsigned€int€**processor_set_info_count*)

ARGUMENTS *processor_set*: The processor set for which information is to be obtained.

flavor: The type of information that is wanted. Should be PROCESSOR_SET_BASIC_INFO or PROCESSOR_SET_SCHED_INFO.

host: Returns the nonprivileged host port for the host on which the processor set resides.

processor_set_info: Returns information about the processor set specified by *processor_set*.

processor_set_info_count: Size of the info structure. Should be PROCESSOR_SET_BASIC_INFO_COUNT when *flavor* is PROCESSOR_SET_BASIC_INFO, and PROCESSOR_SET_SCHED_INFO_COUNT when *flavor* is PROCESSOR_SET_SCHED_INFO.

DESCRIPTION Returns the selected information array for a processor set, as specified by *flavor*. The *processor_set_info* argument is an array of integers that is supplied by the caller, and filled with specified information. The *processor_set_info_count* argument is supplied as the maximum number of integers in *processor_set_info*. On return, it contains the actual number of integers in *processor_set_info*.

Basic information is defined by PROCESSOR_SET_BASIC_INFO. The size of this information is defined by PROCESSOR_SET_BASIC_INFO_COUNT. The **load_average** and **mach_factor** fields are scaled by the constant LOAD_SCALE (that is,€the integer value returned is the load average or Mach factor multiplied by LOAD_SCALE).

The *Mach factor*, like the UNIX load average, is a measurement of how busy the system is. Unlike the load average, higher Mach factors mean that the system is less busy. The Mach factor tells you how much of a CPU you have available for running an application. For example, on a single-processor system with one job running, the Mach factor is 0.5; this means if another job starts running it will get half of the CPU. (Two jobs will be running, each getting half the CPU.) On a single-processor system, the Mach factor is between zero and one. On a multiprocessor system, the Mach factor can go over one. For example, a three-processor system with one job running has a Mach factor of 2.0, since two processors are available to new jobs.

```
struct processor_set_basic_info {
    int  processor_count;   /* number of processors */
    int  task_count;        /* number of tasks */
    int  thread_count;      /* number of threads */
    int  load_average;      /* scaled load average */
    int  mach_factor;       /* scaled mach factor */
};
typedef struct processor_set_basic_info *processor_set_basic_info_t;
```

Scheduling information is defined by PROCESSOR_SET_SCHED_INFO. The size of this information is given by PROCESSOR_SET_SCHED_INFO_COUNT.

```
struct processor_set_sched_info {
    int  policies;          /* allowed policies */
    int  max_priority;      /* max priority for new threads */
};
typedef struct processor_set_sched_info *processor_set_sched_info_t;
```

EXAMPLE

```
kern_return_t                      error;
host_t                             host;
unsigned int                       info_count;
struct processor_set_basic_info    info;
processor_set_t                    default_set;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS){
    mach_error("Error calling processor_set_default", error);
    exit(1);
}

info_count=PROCESSOR_SET_BASIC_INFO_COUNT;
error=processor_set_info(default_set, PROCESSOR_SET_BASIC_INFO,
   &host, (processor_set_info_t)&info, &info_count);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_set_info", error);

printf("The UNIX load average is %f\n",
    (float)info.load_average/LOAD_SCALE);
printf("The Mach factor is %f\n", (float)info.mach_factor/LOAD_SCALE);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor_set* is not a processor set, or *flavor* is not€recognized.

KERN_FAILURE: *processor_set_info_count* is less than what it should be.

MIG_ARRAY_TOO_LARGE: Returned info array is too large for *processor_set_info*.

SEE ALSO **processor_set_create()**, **processor_set_default()**, **processor_assign()**, **task_assign()**, **thread_assign()**

processor_set_max_priority()

SUMMARY Set the maximum priority permitted on a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_max_priority(**processor_set_t *processor_set*, int€*max_priority*, boolean_t *change_threads***)**

DESCRIPTION This function affects only newly created or newly assigned threads unless you specify *change_threads* as true.

Note: This function is useful only on multiprocessor systems.

processor_set_policy_enable(), processor_set_policy_disable()

SUMMARY Enable or disable a scheduling policy on a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_policy_enable(**processor_set_t *processor_set*, int€*policy***)**
kern_return_t **processor_set_policy_disable(**processor_set_t *processor_set*, int€*policy*, boolean_t *change_threads***)**

ARGUMENTS *processor_set*: The processor set whose allowed policies are to be changed. This must be the privileged processor set port, which is returned by **host_processor_set_priv()**.

policy: The policy to enable or disable. Currently, the only valid policies are POLICY_TIMESHARE, POLICY_INTERACTIVE, and POLICY_FIXEDPRI. You can't disable timesharing.

change_threads: Specify true if you want to reset to timesharing the policies of any threads with the newly disallowed policy. Otherwise, specify false.

DESCRIPTION Processor sets may restrict the scheduling policies to be used for threads assigned to them. These two calls provide the mechanism for designating permitted and forbidden policies. The current set of permitted policies can be obtained from **processor_set_info()**. Timesharing can't be forbidden by any processor set. This is a compromise to reduce the complexity of the assign operation; any thread whose policy is forbidden by the target processor set has its policy reset to timesharing. If the *change_threads* argument to **processor_set_policy_disable()** is true, threads currently assigned to this processor set and using the newly disabled policy will have their policy reset to timesharing.

Warning: Don't use POLICY_FIXEDPRI unless you're familiar with the consequences of€fixed-priority scheduling. Using fixed-priority scheduling in a process can keep other processes from getting any CPU time. If processes that are essential to the functioning of€the system don't get CPU time, you might have to reboot your system to make it work€normally.

EXAMPLE

```
kern_return_t    error;
processor_set_t  default_set, default_set_priv;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}
```

```
error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

error=processor_set_policy_enable(default_set_priv, POLICY_FIXEDPRI);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_set_policy_enable", error);
```

RETURN KERN_SUCCESS: Operation completed successfully.

KERN_INVALID_ARGUMENT: *processor_set* isn't the privileged port of a processor set, *policy* isn't a valid policy, or an attempt was made to disable timesharing.

SEE ALSO **thread_policy()**, **thread_switch()**

processor_set_tasks()

SUMMARY Get kernel ports for tasks assigned to a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_tasks(**processor_set_t *processor_set*, task_array_t€**task_list*, unsigned int **task_count***)**

ARGUMENTS *processor_set*: The processor set to be affected. This must be the privileged processor set port, which is returned by **host_processor_set_priv()**.

task_list: Returns the set of tasks currently assigned to *processor_set*; no particular ordering is guaranteed.

task_count: Returns the number of tasks in *task_list*.

DESCRIPTION This function gets send rights to the kernel port for each task currently assigned to *processor_set*. The *task_list* argument is an array that is created as a result of this call. You€should call **vm_deallocate()** on this array when you no longer need the data.

EXAMPLE

```
task_array_t    task_list;
unsigned int    task_count;
processor_set_t default_set, default_set_priv;
kern_return_t   error;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

error=processor_set_tasks(default_set_priv, &task_list, &task_count);
if (error != KERN_SUCCESS) {
    mach_error("Call to processor_set_tasks() failed", error);
    exit(1);
}
```

```
/* . . . */
error=vm_deallocate(task_self(), (vm_address_t)task_list,
    sizeof(task_list)*task_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing task_list", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor_set* isn't a privileged processor set.

SEE ALSO **task_assign()**, **thread_assign()**, **processor_set_threads()**

processor_set_threads()

SUMMARY Get kernel ports for threads assigned to a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_threads(**processor_set_t *processor_set*, thread_array_t€**thread_list*, unsigned int **thread_count***)**

ARGUMENTS *processor_set*: The processor set to be affected. This must be the privileged processor set port, which is returned by **host_processor_set_priv()**.

thread_list: Returns the set of threads currently assigned to *processor_set*; no particular ordering is guaranteed.

thread_count: Returns the number of threads in *thread_list*.

DESCRIPTION This function gets send rights to the kernel port for each thread currently assigned to *processor_set*. The *thread_list* argument is an array that is created as a result of this call. You should call **vm_deallocate()** on *thread_list* when you no longer need the data.

EXAMPLE
```
thread_array_t  thread_list;
unsigned int    thread_count;
processor_set_t default_set, default_set_priv;
kern_return_t   error;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

error=processor_set_threads(default_set_priv, &thread_list, &thread_count);
if (error != KERN_SUCCESS) {
    mach_error("Call to processor_set_threads() failed", error);
    exit(1);
}

/* . . . */
error=vm_deallocate(task_self(), (vm_address_t)thread_list,
    sizeof(thread_list)*thread_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing thread_list", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor_set* isn't a privileged processor set.

SEE ALSO **task_assign()**, **thread_assign()**, **processor_set_tasks()**

task_assign(), task_assign_default()

SUMMARY Assign a task to a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_assign(**task_t *task*, processor_set_t *new_processor_set*, boolean_t€*assign_threads***)**
kern_return_t **task_assign_default(**task_t *task*, boolean_t *assign_threads***)**

DESCRIPTION The **task_assign()** function assigns *task* to *new_processor_set*; **task_assign_default()** assigns *task* to the default processor set.

Note: These functions are useful only on multiprocessor systems.

task_by_unix_pid()

SUMMARY Get the task port for a UNIX process on the same host

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_by_unix_pid(**task_t *task*, int *pid*, task_t **result_task***)**

ARGUMENTS *task*: A task that is used to check permission (usually **task_self()**).

pid: The process ID of the desired process.

result_task: Returns send rights to the task port of the process specified by *pid*.

DESCRIPTION Returns the task port for another process, named by its process ID, on the same host as *task*. This call succeeds only if the caller is the superuser or *task* has the same user ID as the process specified by *pid*. If the call fails, *result_task* is set to TASK_NULL.

EXAMPLE

```
pid=fork();

if (pid==0) /* We're in the child. */ {
    /* do childish things */
}
else /* We're in the parent */ {
    result=task_by_unix_pid(task_self(), pid, &child_task);
    if (result != KERN_SUCCESS)
        mach_error("task_by_unix_pid", result);
    /* . . . */
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_FAILURE: *target_task* has a different user ID from the process corresponding to *pid,* and the caller isn't the superuser; or *pid* didn't refer to a valid process; or *target_task* wasn't a valid task.

SEE ALSO **unix_pid()**

task_create()

SUMMARY Create a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_create(**task_t *parent_task*, boolean_t *inherit_memory*, task_t€**child_task***)**

ARGUMENTS *parent_task*: The task from which the child's capabilities are drawn.

inherit_memory: If set, the address space of the child task is built from the parent task according to its memory inheritance values; otherwise, the child task is given an empty address space.

child_task: Returns the new task.

DESCRIPTION The function **task_create()** creates a new task from *parent_task*; the resulting task (*child_task*) acquires shared or copied parts of the parent's address space (see **vm_inherit()**). The child task initially has no threads; you put threads in it using **thread_create()**.

Important: Normally, you should use the UNIX **fork()** system call instead of **task_create()**.

The child task gets the four special ports initialized for it at task creation. The kernel port (task port) is created, and send rights for it are given to the child and returned to the caller in *child_task*. The notify port is initialized to null. The child inherits its bootstrap and exception ports from the parent task. The new task can get send rights to these ports with the call **task_get_special_port()**.

EXAMPLE
```
error=task_create(task_self(), TRUE, &child_task);
if(error!=KERN_SUCCESS)
    mach_error("Call to task_create() failed", error);
```

RETURN KERN_SUCCESS: A new task has been created.

KERN_INVALID_ARGUMENT: *parent_task* is not a valid task port.

KERN_RESOURCE_SHORTAGE: Some critical kernel resource is unavailable.

SEE ALSO **task_terminate()**, **task_suspend()**, **task_resume()**, **task_get_special_port()**, **task_set_special_port()**, **task_self()**, **task_threads()**, **thread_create()**, **thread_resume()**, **vm_inherit()**

task_get_assignment()

SUMMARY Get the name of the processor set that a task is assigned to

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_get_assignment(**task_t *task*, processor_set_t **processor_set***)**

Note: This function is useful only on multiprocessor systems.

task_get_special_port(), task_set_special_port(), task_self(), task_notify()

SUMMARY Get or set a task's special ports

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_get_special_port(**task_t *task*, int *which_port*, port_t€**special_port***)**
kern_return_t **task_set_special_port(**task_t *task*, int *which_port*, port_t *special_port***)**
task_t **task_self(**void**)**
port_t **task_notify(**void**)**

ARGUMENTS *task*: The task to get the port for.

which_port: The port that's requested. This is one of:

TASK_NOTIFY_PORT
TASK_BOOTSTRAP_PORT
TASK_EXCEPTION_PORT

special_port: The value of the port that's being requested or set.

DESCRIPTION The function **task_get_special_port()** returns send rights to one of a set of special ports for€the task specified by *task.* In the case of the task's own notify port, the task also gets receive rights.

The function **task_set_special_port()** sets one of a set of special ports for the task specified by *task*.

The function **task_self()** returns the port to which kernel calls for the currently executing thread should be directed. Currently, **task_self()** returns the task kernel port, which is a port for which the kernel has receive rights and which it uses to identify a task. In the future it may be possible for one task to interpose a port as another task's kernel port. At that time **task_self()** will still return the port to which the executing thread should direct kernel calls, but it may no longer be a port for which the kernel has receive rights.

If a controller task has send access to the kernel port of a subject task, then the controller task can perform kernel operations for the subject task. Normally, only the task itself and the task that created it will have access to the task kernel port, but any task may pass rights to its kernel port to any other task.

The function **task_notify()** returns receive and send rights to the notify port associated with the task to which the executing thread belongs. The notify port is a port on which the task should receive notification of such kernel events as the destruction of a port to which it has send rights.

The other special ports associated with a task are the bootstrap port and the exception port. The bootstrap port is a port to which a thread may send a message requesting other system service ports. This port isn't used by the kernel. The task's exception port is the port to which messages are sent by the kernel when an exception occurs and the thread causing the exception has no exception port of its own.

Important: If you set your task's bootstrap port, you should also set the global variable **bootstrap_port** to *special_port*. The **bootstrap_port** variable is task-wide and is used by **mach_init** and other processes to determine your task's bootstrap port. Since you can't change the value of the **bootstrap_port** variable in another task, you should use care when changing the bootstrap port of another task.

MACRO EQUIVALENTS The following macros are defined in the header file **mach/task_special_ports.h**:

task_get_notify_port(*task*, *port***)**
task_set_notify_port(*task*, *port***)**

task_get_exception_port(*task*, *port***)**
task_set_exception_port(*task*, *port***)**

task_get_bootstrap_port(*task*, *port***)**
task_set_bootstrap_port(*task*, *port***)**

EXAMPLE

```
/* Save the old exception port for this task. */
r = task_get_exception_port(task_self(), &(ports.old_exc_port));
if (r != KERN_SUCCESS) {
    mach_error("task_get_exception_port", r);
    exit(1);
}
```

```
    /* Create a new exception port for this task. */
r = port_allocate(task_self(), &(ports.exc_port));
if (r != KERN_SUCCESS) {
    mach_error("port_allocate 0", r);
    exit(1);
}
r = task_set_exception_port(task_self(), (ports.exc_port));
if (r != KERN_SUCCESS) {
    mach_error("task_set_exception_port", r);
    exit(1);
}
```

RETURN KERN_SUCCESS: The port was returned or set.

KERN_INVALID_ARGUMENT: Either *task* is not a task or *which_port* is an invalid port selector.

SEE ALSO **thread_special_ports()**, **task_create()**

task_info()

SUMMARY Get information about a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_info(**task_t *target_task*, int *flavor*, task_info_t *task_info*, unsigned€int€**task_info_count***)**

ARGUMENTS *target_task*: The task to be affected (for example, use **task_self()** to specify the caller's€task).

flavor: The type of statistics that are wanted. Currently only TASK_BASIC_INFO is€implemented.

task_info: Returns statistics about *target_task*.

task_info_count: Size of the info structure. Currently this must be TASK_BASIC_INFO_COUNT.

DESCRIPTION The function **task_info()** returns the information specified by *flavor* about a task. The *task_info* argument is an array of integers that's supplied by the caller and returned filled with information. The *task_info_count* argument is supplied as the maximum number of integers in *task_info.* On return, it contains the actual number of integers in *task_info*.

Currently there's only one flavor of information, defined by TASK_BASIC_INFO. Its size is defined by TASK_BASIC_INFO_COUNT. The definition of the information structure returned by TASK_BASIC_INFO is:

```
struct task_basic_info {
    int           suspend_count;  /* suspend count for task */
    int           base_priority;  /* base scheduling priority */
    vm_size_t     virtual_size;   /* number of virtual pages */
    vm_size_t     resident_size;  /* number of resident pages */
    time_value_t  user_time;      /* total user run time for
                                     terminated threads */
    time_value_t  system_time;    /* total system run time for
                                     terminated threads */
};
typedef struct task_basic_info  *task_basic_info_t;
```

EXAMPLE

```
kern_return_t            error;
struct task_basic_info    info;
unsigned int              info_count=TASK_BASIC_INFO_COUNT;

error=task_info(task_self(), TASK_BASIC_INFO,
    (task_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling task_info()", error);
```

```
else
    printf("Base priority is %d\n", info.base_priority);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *target_task* isn't a task, or *flavor* isn't recognized.

MIG_ARRAY_TOO_LARGE: The returned info array is too large for *task_info*. The *task_info* argument is filled as much as possible, and *task_info_count* is set to the number of elements that would be returned if there were enough room.

SEE ALSO **task_threads()**, **thread_info()**, **thread_get_state()**

task_priority()

SUMMARY Set the scheduling priority for a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_priority(**task_t *task*, int *priority*, boolean_t *change_threads***)**

ARGUMENTS *task*: Task to set priority for.

priority: New priority.

change_threads: Change priority of existing threads if true.

DESCRIPTION The priority of a task is used only for creation of new threads; the priority of a new thread priority is set to that of its task. The **task_priority()** function changes this task priority. It also sets the priorities of all threads in the task to this new priority if *change_threads* is true. Existing threads are not affected otherwise. If this priority change violates the maximum priority of some threads, as many threads as possible will be changed and an error code will be returned.

Priorities range from 0 to 31, where higher numbers denote higher priorities. You can retrieve the current scheduling priority using **thread_info()**.

EXAMPLE

```
kern_return_t          error;
struct task_basic_info    info;
unsigned int              info_count=TASK_BASIC_INFO_COUNT;

error=task_info(task_self(), TASK_BASIC_INFO,
    (task_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling task_info()", error);
else {
    /* Set this task's base priority to be much lower than normal */
    error = task_priority(task_self(), info.base_priority - 4, TRUE);
    if (error != KERN_SUCCESS)
        mach_error("Call to task_priority() failed", error);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* is not a task, or *priority* is not a valid priority.

KERN_FAILURE: *change_threads* was true and the attempt to change the priority of at least one existing thread failed because the new priority would have exceeded that thread's maximum priority.

SEE ALSO **thread_priority()**, **processor_set_max_priority()**, **thread_switch()**

task_resume()

SUMMARY Resume a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_resume(**task_t *target_task***)**

ARGUMENTS *target_task*: The task to be resumed.

DESCRIPTION The function **task_resume()** decrements the task's suspend count. If the suspend count becomes 0, all threads with 0 suspend counts in the task are resumed. If the suspend count is already 0, it's not decremented (it never becomes negative).

RETURN KERN_SUCCESS: The task has been resumed.

KERN_FAILURE: The suspend count is already 0.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO **task_create()**, **task_terminate()**, **task_suspend()**, **task_info()**, **thread_suspend()**, **thread_resume()**, **thread_info()**

task_suspend()

SUMMARY Suspend a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_suspend(**task_t *target_task***)**

ARGUMENTS *target_task*: The task to be suspended (for example, use **task_self()** to specify the caller's€task).

DESCRIPTION The function **task_suspend()** increments the task's suspend count and stops all threads in the task. As long as the suspend count is positive, newly created threads will not run. This call doesn't return until all threads are suspended.

If the count becomes greater than 1, it will take more than one **task_resume()** call to restart the task.

RETURN KERN_SUCCESS: The task has been suspended.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO **task_create()**, **task_terminate()**, **task_resume()**, **task_info()**, **thread_suspend()**

task_terminate()

SUMMARY Terminate a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_terminate(**task_t *target_task***)**

ARGUMENTS *target_task*: The task to be destroyed (for example, use **task_self()** to specify the caller's€task).

DESCRIPTION The function **task_terminate()** destroys the task specified by *target_task* and all its threads. All resources that are used only by this task are freed. Any port to which this task has receive rights is destroyed.

RETURN KERN_SUCCESS: The task has been destroyed.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO **task_create()**, **task_suspend()**, **task_resume()**, **thread_terminate()**, **thread_suspend()**

task_threads()

SUMMARY Get a task's threads

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_threads(**task_t *target_task*, thread_array_t **thread_list*, unsigned€int€**thread_count***)**

ARGUMENTS *target_task*: The task to be affected (for example, use **task_self()** to specify the caller's€task).

thread_list: Returns the set of threads contained within *target_task*; no particular ordering is guaranteed.

thread_count: Returns the number of threads in *thread_list*.

DESCRIPTION The function **task_threads()** gets send rights to the kernel port for each thread contained in *target_task*.

The array *thread_list* is created as a result of this call. You should call **vm_deallocate()** on this array when the data is no longer needed.

EXAMPLE
```
r = task_threads(task_self(), &thread_list, &thread_count);
if (r != KERN_SUCCESS)
    mach_error("Error calling task_threads", r);
else {
    if (thread_count == 1)
        printf ("There's 1 thread in this task\n");
    else
        printf("There are %d threads in this task\n", thread_count);

/* Deallocate the list of threads. */
    r = vm_deallocate(task_self(), (vm_address_t)thread_list,
        sizeof(thread_list)*thread_count);
    if (r != KERN_SUCCESS)
        mach_error("Trouble freeing thread_list", r);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO **thread_create()**, **thread_terminate()**, **thread_suspend()**

thread_abort()

SUMMARY Interrupt a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_abort(**thread_t *target_thread***)**

ARGUMENTS *target_thread*: The thread to be interrupted.

DESCRIPTION The function **thread_abort()** aborts page faults and the kernel functions **msg_send()**, **msg_receive()**, and **msg_rpc()**, making the call return a code indicating that it was interrupted. The call is interrupted whether or not the thread (or task containing it) is currently suspended. If it's suspended, the thread receives the interrupt when it resumes.

A thread will retry an aborted page fault if its state isn't modified before it resumes. The function **msg_send()** returns SEND_INTERRUPTED; **msg_receive()** returns RCV_INTERRUPTED; and **msg_rpc()** returns either SEND_INTERRUPTED or RCV_INTERRUPTED, depending on which half of the RPC was interrupted.

This function lets one thread stop another thread cleanly, thereby allowing the future execution of the target thread to be controlled in a predictable way. The **thread_suspend()** function keeps the target thread from executing any further instructions at the user level, including the return from a system call. The **thread_get_state()** and **thread_set_state()** functions let you examine or modify the user state of a target thread. However, if a suspended thread was executing within a system call, it also has associated with it a kernel state. This kernel state can't be modified by **thread_set_state()**; therefore, when the thread is resumed the system call may return, changing the user state and possibly user memory.

The **thread_abort()** function aborts the kernel call from the target thread's point of view by resetting the kernel state so that the thread will resume execution just after the system call. The system call will return one of the interrupted codes described previously. The system call will either be entirely completed or entirely aborted, depending on the precise moment at which the abort was received. Thus if the thread's user state has been changed by **thread_set_state()**, it won't be modified by any unexpected system call side effects.

For example, to simulate a UNIX signal, the following sequence of calls may be used:

1. **thread_suspend()**ÐStops the thread.
2. **thread_abort()**ÐInterrupts any system call in progress, setting the return value to ªinterrupted.º Since the thread is stopped, it won't return to user code.
3. **thread_set_state()**ÐAlters the thread's state to simulate a procedure call to the signal handler.
4. **thread_resume()**ÐResumes execution at the signal handler. If the thread's stack has been correctly set up, the thread can return to the interrupted system call.

Calling **thread_abort()** on a thread that's not suspended is risky, since it's difficult to know exactly what system trap, if any, the thread might be executing and whether an interrupt return would cause the thread to do something useful.

RETURN KERN_SUCCESS: The thread received an interrupt.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO **thread_get_state()**, **thread_info()**, **thread_terminate()**, **thread_suspend()**

thread_assign(), thread_assign_default()

SUMMARY Assign a thread to a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_assign(**thread_t *thread*, processor_set_t *new_processor_set***)**
kern_return_t **thread_assign_default(**thread_t *thread***)**

DESCRIPTION **thread_assign()** assigns *thread* to *new_processor_set*; **thread_assign_default()** assigns *thread* to the default processor set.

Note: These functions are useful only on multiprocessor systems.

thread_create()

SUMMARY Create a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_create(**task_t *parent_task*, thread_t **child_thread***)**

ARGUMENTS *parent_task*: The task that should contain the new thread.

child_thread: Returns the new thread.

DESCRIPTION The function **thread_create()** creates a new thread within *parent_task.* The new thread has€no processor state, and has a suspend count of 1. To get a new thread to run, first call€**thread_create()** to get the new thread's identifier, *child_thread.* Then call **thread_set_state()** to set a processor state. Finally, call **thread_resume()** to schedule the€thread to execute.

Important: Don't use this function unless you're writing a loadable kernel server or implementing a new thread package, such as the C-thread functions. For normal, user-level programming, use **cthread_fork()** instead. You can then use **cthread_thread()** if you need to get the Mach thread that corresponds to the new C-thread.

When the thread is created, send rights to its thread kernel port are given to it and returned to the caller in *child_thread.* The new thread's exception port is set to PORT_NULL.

RETURN KERN_SUCCESS: A new thread has been created.

KERN_INVALID_ARGUMENT: *parent_task* isn't a valid task.

KERN_RESOURCE_SHORTAGE: Some critical kernel resource isn't available.

SEE ALSO **task_create()**, **task_threads()**, **thread_terminate()**, **thread_suspend()**, **thread_resume()**, **thread_special_ports()**, **thread_set_state()**

thread_get_assignment()

SUMMARY Get the name of the processor set to which a thread is assigned

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_get_assignment(**thread_t *thread*, processor_set_t **processor_set***)**

Note: This function is useful only in multiprocessor systems.

thread_get_special_port(), thread_set_special_port(), thread_self(), thread_reply()

SUMMARY Get or set a thread's special ports

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_get_special_port(**thread_t *thread*, int *which_port*, port_t€**special_port***)**
kern_return_t **thread_set_special_port(**thread_t *thread*, int *which_port*, port_t€*special_port***)**
thread_t **thread_self(**void**)**

port_t **thread_reply**(void)

ARGUMENTS *thread*: The thread to get the port for.

which_port: The port that's requested. This is one of:

THREAD_REPLY_PORT
THREAD_EXCEPTION_PORT

special_port: The value of the port that's being requested or set.

DESCRIPTION The function **thread_get_special_port()** returns send rights to one of a set of special ports for the thread specified by *thread.* In the case of getting the thread's own reply port, receive rights are also given to the thread.

The function **thread_set_special_port()** sets one of a set of special ports for the thread specified by *thread*.

The function **thread_self()** returns the port to which kernel calls for the currently executing thread should be directed. Currently **thread_self()** returns the thread kernel port, which is a port for which the kernel has receive rights and which it uses to identify a thread. In the future it may be possible for one thread to interpose a port as another thread's kernel port. At that time **thread_self()** will still return the port to which the executing thread should direct kernel calls, but it may no longer be a port for which the kernel has receive rights.

If a controller thread has send access to the kernel port of a subject thread, the controller thread can perform kernel operations for the subject thread. Normally only the thread itself and its parent task will have access to the thread kernel port, but any thread may pass rights to its kernel port to any other thread.

The function **thread_reply()** returns receive and send rights to the reply port of the calling thread. The reply port is a port to which the thread has receive rights. It's used to receive any initialization messages and as a reply port for early remote procedure calls.

A thread also has access to its task's special ports.

MACRO EQUIVALENTS The following macros are defined in the header file **mach/thread_special_ports.h**:

thread_get_reply_port(*thread*, *port***)**
thread_set_reply_port(*thread*, *port***)**

thread_get_exception_port(*thread*, *port***)**
thread_set_exception_port(*thread*, *port***)**

RETURN KERN_SUCCESS: The port was returned or set.

KERN_INVALID_ARGUMENT: *thread* isn't a thread, or *which_port* is an invalid port€selector.

SEE ALSO **task_get_special_port()**, **task_set_special_port()**, **task_self()**, **thread_create()**

thread_get_state(), thread_set_state()

SUMMARY Get or set a thread's state

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_get_state(**thread_t *target_thread*, int *flavor*, thread_state_data_t€*old_state*, unsigned int **old_state_count***)**
kern_return_t **thread_set_state(**thread_t *target_thread*, int *flavor*, thread_state_data_t€*new_state*, unsigned int *new_state_count***)**

ARGUMENTS *target_thread*: The thread whose state is affected.

flavor: The type of state that's to be manipulated. This may be any one of the following:

NeXT_THREAD_STATE_REGS
NeXT_THREAD_STATE_68882
NeXT_THREAD_STATE_USER_REG

old_state: Returns an array of state information.

new_state: An array of state information.

old_state_count: The size of the state information array. This may be any one of the following:

NeXT_THREAD_STATE_REGS_COUNT
NeXT_THREAD_STATE_68882_COUNT
NeXT_THREAD_STATE_USER_REG_COUNT

new_state_count: Same as *old_state_count*.

DESCRIPTION The function **thread_get_state()** returns the state component (that is, the machine registers) of *target_thread* as specified by *flavor*. The *old_state* is an array of integers that's€provided by the caller and returned filled with the specified information. You should€set *old_state_count* to the maximum number of integers in *old_state*. On return, *old_state_count* is equal to the actual number of integers in *old_state*.

The function **thread_set_state()** sets the state component of *target_thread* as specified by *flavor*. The *new_state* is an array of integers that the caller fills. You should set *new_state_count* to the number of elements in *new_state*. The entire set of registers is reset.

target_thread must not be **thread_self()** for either of these calls.

The state structures are defined in the header file **mach/machine/thread_status.h**.

RETURN KERN_SUCCESS: The state has been set or returned.

MIG_ARRAY_TOO_LARGE: The returned state is too large for the *new_state*. The *new_state* argument is filled in as much as possible, and *new_state_count* is set to the number of elements that would be returned if there were enough room.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread, *target_thread* is **thread_self()**, or *flavor* is unrecognized for this machine.

SEE ALSO **task_info()**, **thread_info()**

thread_info()

SUMMARY Get information about a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_info(**thread_t *target_thread*, int *flavor*, thread_info_t€*thread_info*, unsigned int **thread_info_count***)**

ARGUMENTS *target_thread*: The thread to be affected.

flavor: The type of statistics wanted. This can be THREAD_BASIC_INFO or THREAD_SCHED_INFO.

thread_info: Returns statistics about *target_thread*.

thread_info_count: Size of the info structure. This can be THREAD_BASIC_INFO_COUNT or THREAD_SCHED_INFO_COUNT.

DESCRIPTION The function **thread_info()** returns the selected information array for a thread, as specified by *flavor*. The *thread_info* argument is an array of integers that's supplied by the caller and returned filled with specified information. The *thread_info_count* argument is supplied as the maximum number of integers in *thread_info*. On return, it contains the actual number of integers in *thread_info*.

The size of the information returned by THREAD_BASIC_INFO is defined by THREAD_BASIC_INFO_COUNT. The definition of the information structure returned by THREAD_BASIC_INFO is:

```
struct thread_basic_info {
    time_value_t  user_time;     /* user run time */
    time_value_t  system_time;   /* system run time */
    int           cpu_usage;     /* scaled cpu usage percentage */
    int           base_priority; /* base scheduling priority */
    int           cur_priority;  /* current scheduling priority */
    int           run_state;     /* run state */
    int           flags;         /* various flags */
    int           suspend_count; /* suspend count for thread */
    long          sleep_time;    /* number of seconds that thread
                                    has been sleeping */
};
typedef struct thread_basic_info *thread_basic_info_t;
```

The **run_state** field has one of the following values:

TH_STATE_RUNNING: The thread is running normally.

TH_STATE_STOPPED: The thread is suspended. This happens when the thread or task suspend count is greater than zero.

TH_STATE_WAITING: The thread is sleeping normally.

TH_STATE_UNINTERRUPTIBLE: The thread is in an uninterruptible sleep. This should happen only for very short times during some system calls.

TH_STATE_HALTED: The thread is halted at a clean point. This state happens only after a call to **thread_abort()**.

Possible values of the **flags** field are:

TH_FLAGS_SWAPPED: The thread is swapped out. This happens when the thread hasn't run in a long time, and the kernel stack for the thread has been swapped out.

TH_FLAGS_IDLE: The thread is the idle thread for the CPU. This means that the CPU runs this thread whenever it has no other threads to run.

The **sleep_time** field is useful only when **run_state** is TH_STATE_STOPPED. (Currently **sleep_time** is always set to zero, no matter how long the thread has been sleeping.)

The size of the information returned by THREAD_SCHED_INFO is defined by THREAD_SCHED_INFO_COUNT. The definition of the information structure returned by THREAD_SCHED_INFO is:

```
struct thread_sched_info {
    int       policy;           /* scheduling policy */
    int       data;             /* associated data */
    int       base_priority;    /* base priority */
    int       max_priority;     /* max priority */
    int       cur_priority;     /* current priority */
    boolean_t depressed;        /* depressed ? */
    int       depress_priority; /* priority depressed from */
};
typedef struct thread_sched_info    *thread_sched_info_t;
```

The **policy** field has one of the following values: POLICY_FIXEDPRI, POLICY_TIMESHARE, or POLICY_INTERACTIVE. If **policy** is POLICY_FIXEDPRI, then **data** is the quantum (in milliseconds).

Otherwise, **data** is meaningless.

EXAMPLE Example of using THREAD_BASIC_INFO:

```
kern_return_t             error;
struct thread_basic_info  info;
unsigned int              info_count=THREAD_BASIC_INFO_COUNT;

error=thread_info(thread_self(), THREAD_BASIC_INFO,
    (thread_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling thread_info()", error);
else {
    printf("User time is %d seconds, %d microseconds\n",
        info.user_time.seconds, info.user_time.microseconds);
    printf("System time is %d seconds, %d microseconds\n",
        info.system_time.seconds, info.system_time.microseconds);
}
```

Example of using THREAD_SCHED_INFO:

```
kern_return_t             error;
struct thread_sched_info  info;
unsigned int              info_count=THREAD_SCHED_INFO_COUNT;

error=thread_info(thread_self(), THREAD_SCHED_INFO,
    (thread_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling thread_info()", error);
else {
    printf("Base priority is %d\n", info.base_priority);
    printf("Max priority is %d\n", info.max_priority);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread, or *flavor* isn't recognized, or *thread_info_count* is smaller than it's supposed to be.

MIG_ARRAY_TOO_LARGE: The returned info array is too large for *thread_info.* The *thread_info* argument is filled as much as possible, and *thread_info_count* is set to the number of elements that would have been returned if there were enough room.

SEE ALSO **thread_get_special_port()**, **task_threads()**, **task_info()**, **thread_get_state()**

thread_policy()

SUMMARY Set scheduling policy for a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_policy(**thread_t *thread*, int *policy*, int *data***)**

ARGUMENTS *thread*: Thread to set policy for.

policy: Policy to set. This must be POLICY_TIMESHARE, POLICY_INTERACTIVE, or POLICY_FIXEDPRI.

data: Policy-specific data.

DESCRIPTION This function changes the scheduling policy for *thread* to *policy*.

The *data* argument is meaningless for the timesharing and interactive policies; for the fixed-priority policy, it's the quantum to be used (in milliseconds). The system will always€round the quantum up to the next multiple of the basic system quantum (**min_quantum**, which can be obtained from **host_info()**). You can find the current quantum using **thread_info()**.

Processor sets can restrict the allowed policies, so this call will fail if the processor set to€which *thread* is currently assigned doesn't permit *policy*.

EXAMPLE

```
kern_return_t           error;
struct host_sched_info  sched_info;
unsigned int            sched_count=HOST_SCHED_INFO_COUNT;
int                     quantum;
processor_set_t         default_set, default_set_priv;

/* Set quantum to a reasonable value. */
error=host_info(host_self(), HOST_SCHED_INFO,
    (host_info_t)&sched_info, &sched_count);
if (error != KERN_SUCCESS) {
    mach_error("SCHED host_info() call failed", error);
    exit(1);
}
else
    quantum = sched_info.min_quantum;

/*
 * Fix the default processor set to take a fixed priority thread.
 */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

error=processor_set_policy_enable(default_set_priv, POLICY_FIXEDPRI);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_set_policy_enable", error);

/*
 * Change the thread's scheduling policy to fixed priority.
 */
error=thread_policy(thread_self(), POLICY_FIXEDPRI, quantum);
if (error != KERN_SUCCESS)
    mach_error("thread_policy() call failed", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *thread* is not a thread, or *policy* is not a recognized policy.

KERN_FAILURE: The processor set to which *thread* is currently assigned doesn't permit *policy*.

SEE ALSO **processor_set_policy(), thread_switch()**

thread_priority(), thread_max_priority()

SUMMARY Set scheduling priority for thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_priority(**thread_t *thread*, int *priority*, boolean_t *set_max***)**
kern_return_t **thread_max_priority(**thread_t *thread*, processor_set_t *processor_set*, int€*priority***)**

ARGUMENTS *thread*: The thread whose priority is to be changed.

priority: The new priority to change it to.

set_max: Also set *thread*'s maximum priority if true.

processor_set: The privileged port for the processor set to which *thread* is currently assigned.

DESCRIPTION Threads have three priorities associated with them by the system: a *base priority*, a *maximum priority*, and a *scheduled priority*.

The scheduled priority is used to make scheduling decisions about the thread. It's determined from the base priority by the policy. (For the timesharing and interactive policies, this means adding an increment derived from CPU usage). The base priority can€be set under user control, but can never exceed the maximum priority. Raising the maximum priority requires presentation of the privileged port for the thread's processor set;€since the privileged port for the default processor set is available only to the superuser, users cannot raise their maximum priority to unfairly compete with other users on that set. Newly created threads obtain their base priority from the task and their maximum priority from the thread.

Priorities range from 0 to 31, where higher numbers denote higher priorities. You can obtain the base, scheduled, and maximum priorities using **thread_info()**.

The **thread_priority()** function changes the base priority and optionally the maximum priority of *thread*. If the new base priority is higher than the scheduled priority of the currently executing thread, preemption may occur as a result of this call. The maximum priority of the thread is also set if *set_max* is true. This call fails if *priority* is greater than the current maximum priority of the thread. As a result, **thread_priority()** can lowerÐbut never raiseÐthe value of a thread's maximum priority.

The **thread_max_priority()** function changes the maximum priority of the thread. Because it requires the privileged port for the processor set, this call can reset the maximum priority to any legal value. If the new maximum priority is less than the thread's base priority, then the thread's base priority is set to the new maximum priority.

EXAMPLE

```
/* Get the privileged port for the default processor set. */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error!=KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

/* Set the max priority. */
error=thread_max_priority(thread_self(), default_set_priv, priority);
if (error!=KERN_SUCCESS)
    mach_error("Call to thread_max_priority() failed",error);

/* Set the thread's priority. */
error=thread_priority(thread_self(), priority, FALSE);
if (error!=KERN_SUCCESS)
    mach_error("Call to thread_priority() failed",error);
```

RETURN KERN_SUCCESS: Operation completed successfully.

KERN_INVALID_ARGUMENT: *thread* is not a thread, *processor_set* is not a privileged port for a processor set, or *priority* is out of range (not in 0-31).

KERN_FAILURE: The requested operation would violate the thread's maximum (only for **thread_priority()**), or the thread is not assigned to the processor set whose privileged port was presented.

SEE ALSO **thread_policy()**, **task_priority()**, **processor_set_priority()**, **thread_switch()**

thread_resume()

SUMMARY Resume a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_resume(**thread_t *target_thread***)**

ARGUMENTS *target_thread*: The thread to be resumed.

DESCRIPTION The function **thread_resume()** decrements the thread's suspend count. If the count becomes 0, the thread is resumed. If it's still positive, the thread is left suspended. The suspend count never becomes negative.

RETURN KERN_SUCCESS: The thread has been resumed.

KERN_FAILURE: The suspend count is already 0.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO **task_suspend()**, **task_resume()**, **thread_info()**, **thread_create()**, **thread_terminate()**, **thread_suspend()**

thread_suspend()

SUMMARY Suspend a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_suspend(**thread_t *target_thread***)**

ARGUMENTS *target_thread*: The thread to be suspended.

DESCRIPTION The function **thread_suspend()** increments the thread's suspend count and prevents the thread from executing any more user-level instructions. In this context, a user-level instruction is either a machine instruction executed in user mode or a system trap instruction (including page faults).

If a thread is currently executing within a system trap, the kernel code may continue to execute until it reaches the system return code, or it may suspend within the kernel code. In either case, when the thread is resumed the system trap will return. This could cause unpredictable results if you did a suspend and then altered the user state of the thread in order to change its direction upon a resume. The function **thread_abort()** lets you abort any currently executing system call in a predictable way.

If the suspend count becomes greater than 1, it will take more than one **thread_resume()** call to restart the thread.

RETURN KERN_SUCCESS: The thread has been suspended.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO **task_suspend()**, **task_resume()**, **thread_get_state()**, **thread_info()**, **thread_resume()**, **thread_terminate()**, **thread_abort()**

thread_switch()

SUMMARY Cause a context switch

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_switch(**thread_t *new_thread*, int *option*, int *time***)**

ARGUMENTS *new_thread*: Thread to switch to. If you specify THREAD_NULL, be sure to specify the *option* argument to be either SWITCH_OPTION_WAIT or SWITCH_OPTION_DEPRESS.

option: Specifies options associated with context switch. Three options are recognized:

SWITCH_OPTION_NONE: No options; the *time* argument is ignored. (You must set *new_thread* to a valid thread.)

SWITCH_OPTION_WAIT: This thread is blocked for the specified time. The block can be aborted by **thread_abort()**.

SWITCH_OPTION_DEPRESS: This thread's priority is depressed to the lowest possible value until one of the following happens: *time* milliseconds pass, this thread is scheduled again, or **thread_abort()** is called on this thread (whichever happens first). Priority depression is independent of operations that change this thread's priority; for example, **thread_priority()** will not abort the depression.

time: Time duration (in milliseconds*)* for options. The minimum time can be obtained as the **min_timeout** value from **host_info()**.

DESCRIPTION This function provides low-level access to the scheduler's context switching code. *new_thread* is a hint that implements handoff scheduling. The operating system will attempt to switch directly to *new_thread* (bypassing the normal logic that selects the next thread to run) if possible. If *new_thread* isn't valid or THREAD_NULL, **thread_switch()** returns an error.

The **thread_switch()** function is often called when the current thread can proceed no farther for some reason; the various options and arguments allow information about this reason to be transmitted to the kernel. The *new_thread* argument (handoff scheduling) is useful when the identity of the thread that must make progress before the current thread runs again is known. The SWITCH_OPTION_WAIT option is used when the amount of time that the current thread must wait before it can do anything useful can be estimated and is fairly long. The SWITCH_OPTION_DEPRESS option is used when the required waiting time is fairly short, especially when the identity of the thread that is being waited for is not known.

Users should beware of calling **thread_switch()** with an invalid *new_thread* (for example, THREAD_NULL) and no *option*. Because the timesharing and interactive schedulers vary the priority of threads based on usage, this may result in a waste of CPU time if the thread that must be run is of lower priority. The use of the SWITCH_OPTION_DEPRESS option in this situation is highly recommended.

When a thread that's depressed is scheduled, it regains its old priority. The code should recheck the conditions to see if it wants to depress again. If **thread_abort()** is called on a depressed thread, the priority of the thread is restored.

Users relying on the preemption semantics of a fixed-priority policy should be aware that **thread_switch()** ignores these semantics; it will run the specified *new_thread* independent of its priority and the priority of any other threads that could be run instead.

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *new_thread* is not a thread, or *option* is not a recognized option.

thread_terminate()

SUMMARY Terminate a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_terminate(**thread_t *target_thread***)**

ARGUMENTS *target_thread*: The thread to be destroyed.

DESCRIPTION The function **thread_terminate()** destroys the thread specified by *target_thread*.

Warning: Don't use this function on threads that were created using the C-thread functions. Each C thread must terminate itself either explicitly, by calling **cthread_exit()**, or implicitly, by returning from its top-level function.

RETURN KERN_SUCCESS: The thread has been destroyed.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO **task_terminate()**, **task_threads()**, **thread_create()**, **thread_resume()**, **thread_suspend()**

unix_pid()

SUMMARY Get the process ID of a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **unix_pid(**task_t *target_task*, int **pid***)**

ARGUMENTS *target_task*: The task for which you want the process ID.

pid: Returns the process ID of *target_task*.

DESCRIPTION Returns the process ID of *target_task*. If the call doesn't succeed, *pid* is set to -1.

EXAMPLE

```
result=unix_pid(task_self(), &my_pid);
if (result!=KERN_SUCCESS) {
    mach_error("Call to unix_pid failed", result);
    exit(1);
}

printf("My process ID is %d\n", my_pid);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_FAILURE: *target_task* isn't a valid task. This might be because *target_task* is a pure Mach task (one created using **task_create()**).

SEE ALSO **task_by_unix_pid()**

vm_allocate()

SUMMARY Allocate virtual memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_allocate(**vm_task_t *target_task*, vm_address_t **address*, vm_size_t€*size*, boolean_t *anywhere***)**

ARGUMENTS *target_task*: Task whose virtual memory is to be affected. Use **task_self()** to allocate memory in the caller's address space.

address: Starting address. If *anywhere* is true, the input value of this address will be ignored, and the space will be allocated wherever it's available. If *anywhere* is false, an attempt is made to allocate virtual memory starting at this virtual address. If this address isn't at the beginning of a virtual page, it gets rounded down so that it is. If there isn't enough space at this address, no memory will be allocated. No matter what the value of *anywhere* is, the address at which memory is actually allocated is returned in *address*.

size: Number of bytes to allocate (rounded up by the system to an integral number of virtual€pages).

anywhere: If true, the kernel should find and allocate any region of the specified size. If€false, virtual memory is allocated starting at *address* (rounded down to a virtual page boundary) if sufficient space exists.

DESCRIPTION The function **vm_allocate()** allocates a region of virtual memory, placing it in the address space of the specified task. The physical memory isn't actually allocated until the new virtual memory is referenced. By default, the kernel rounds all addresses down to the nearest page boundary and all memory sizes up to the nearest page size. The global variable **vm_page_size** contains the page size. For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics()**.

Initially, the pages of allocated memory are protected to allow all forms of access, and are inherited in child tasks as a copy. Subsequent calls to **vm_protect()** and **vm_inherit()** may be used to change these properties. The allocated region is always zero-filled.

Note: Unless you have a special reason for calling **vm_allocate()** (such as a need for page-aligned memory), you should usually call **malloc()** or a similar C library function instead. The C library functions don't necessarily make UNIX or Mach system calls, so they're generally faster than using a Mach function such as **vm_allocate()**.

EXAMPLE

```
if ((ret = vm_allocate(task_self(), (vm_address_t *)&lock,
    sizeof(int), TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate returned value of ", ret);
    printf("Exiting with error.\n");
    exit(-1);
}
if ((ret = vm_inherit(task_self(), (vm_address_t)lock, sizeof(int),
        VM_INHERIT_SHARE)) != KERN_SUCCESS) {
    mach_error("vm_inherit returned value of ", ret);
    printf("Exiting with error.\n");
    exit(-1);
}
```

RETURN KERN_SUCCESS: Memory allocated.

KERN_INVALID_ADDRESS: Illegal address specified.

KERN_NO_SPACE: Not enough space left to satisfy this request.

SEE ALSO **vm_deallocate()**, **vm_inherit()**, **vm_protect()**, **vm_region()**, **vm_statistics()**

vm_copy()

SUMMARY Copy virtual memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_copy(**vm_task_t *target_task*, vm_address_t *source_address*, vm_size_t€*size*, vm_address_t *dest_address*)

ARGUMENTS *target_task*: The task whose virtual memory is to be affected.

source_address: The address in *target_task* of the start of the source range (must be a page boundary).

size: The number of bytes to copy (must be a multiple of **vm_page_size**).

dest_address: The address in *target_task* of the start of the destination range (must be a page boundary).

DESCRIPTION The function **vm_copy()** causes the source memory range to be copied to the destination address; the destination region must not overlap the source region. The destination address range must already be allocated and writable; the source range must be readable.

For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics()**.

EXAMPLE

```
if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data1,
        vm_page_size, TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate returned value of ", rtn);
    printf("vm_copy: Exiting.\n");
    exit(-1);
}

temp = data1;
for (i = 0; (i < vm_page_size / sizeof(int)); i++)
    temp[i] = i;
printf("vm_copy: set data\n");

if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data2,
        vm_page_size, TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate returned value of ", rtn);
    printf("vm_copy: Exiting.\n");
    exit(-1);
}

if ((rtn = vm_copy(task_self(), (vm_address_t)data1, vm_page_size,
        (vm_address_t)data2)) != KERN_SUCCESS) {
    mach_error("vm_copy returned value of ", rtn);
    printf("vm_copy: Exiting.\n");
    exit(-1);
}
```

RETURN KERN_SUCCESS: Memory copied.

KERN_INVALID_ARGUMENT: The address doesn't start on a page boundary or the size isn't a multiple of **vm_page_size**.

KERN_PROTECTION_FAILURE: The destination region isn't writable, or the source region isn't readable.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified, or insufficient memory was allocated at one of the addresses.

SEE ALSO **vm_allocate()**, **vm_protect()**, **vm_write()**, **vm_statistics()**

vm_deactivate()

SUMMARY Mark virtual memory as unlikely to be used soon

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_deactivate(**vm_task_t *target_task*, vm_address_t *address*, vm_size_t€*size*, int€*when***)**

ARGUMENTS *target_task*: Task whose virtual memory is to be affected.

address: Starting address (must be on a page boundary).

size: The number of bytes to deactivate (must be a multiple of **vm_page_size**). Specifying 0 deactivates all of the task's memory at or above *address*.

when: A mask specifying how aggressively the system should deactivate the memory, and whether the memory should be deactivated if it's shared. Values for *when* are defined in the header file **mach/vm_policy.h**.

DESCRIPTION This function lets you tell the operating system that a region of memory won't be used for a long time. It differs from **vm_deallocate()** in that the task's mapping to the memory is retained; only the physical memory associated with the region is affected.

A *when* value of VM_DEACTIVATE_NOW is the most extremeÐthe system will immediately place clean pages at the front of the free list, and dirty pages at the front of the inactive list. A *when* value of VM_DEACTIVATE_SOON specifies that the system should place all pages on the tail of the inactive list. You can add the mask VM_DEACTIVATE_SHARED to indicate that only shared memory should be affected.

This call is used in the window server to deactivate the backing stores of windows in hidden applications, and is used in the Application Kit to deactivate the text, data, and stack of hidden applications.

SEE ALSO **vm_set_policy()**

vm_deallocate()

SUMMARY Deallocate virtual memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_deallocate(**vm_task_t *target_task*, vm_address_t *address*, vm_size_t€*size***)**

ARGUMENTS *target_task*: Task whose virtual memory is to be affected.

address: Starting address (this gets rounded down to a page boundary).

size: Number of bytes to deallocate (this gets rounded up to a page boundary).

DESCRIPTION The function **vm_deallocate()** relinquishes access to a region of a task's address space, causing further access to that memory to fail. This address range will be available for reallocation. Note that because of the rounding to virtual page boundaries, more than *size* bytes may be deallocated. Use **vm_statistics()** or the global variable **vm_page_size** to get the current virtual page size.

This function may be used to deallocate memory that was passed to a task in a message (using out-of-line data). In that case, the rounding should cause no trouble, since the region of memory was allocated as a set of pages.

The function **vm_deallocate()** affects only the task specified by *target_task.* Other tasks that may have access to this memory can continue to reference it.

EXAMPLE

```
r = vm_deallocate(task_self(), (vm_address_t)thread_list,
    sizeof(thread_list)*thread_count);
if (r != KERN_SUCCESS)
    mach_error("Trouble freeing thread_list", r);
```

RETURN KERN_SUCCESS: Memory deallocated.

KERN_INVALID_ADDRESS: Illegal or nonallocated address specified.

SEE ALSO **vm_allocate()**, **vm_statistics()**, **msg_receive()**

vm_inherit()

SUMMARY Inherit virtual memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_inherit(**vm_task_t *target_task*, vm_address_t *address*, vm_size_t€*size*, vm_inherit_t *new_inheritance***)**

ARGUMENTS *target_task*: Task whose virtual memory is to be affected.

address: Starting address (this gets rounded down to a page boundary).

size: Size in bytes of the region for which inheritance is to change (this gets rounded up to a page boundary).

new_inheritance: How this memory is to be inherited in child tasks. Inheritance is specified by using one of these following three values:

VM_INHERIT_SHARE: Child tasks will share this memory with this task.
VM_INHERIT_COPY: Child tasks will receive a copy of this region.
VM_INHERIT_NONE: This region will be absent from child tasks.

DESCRIPTION The function **vm_inherit()** specifies how a region of a task's address space is to be passed to child tasks at the time of task creation. Inheritance is an attribute of virtual pages; thus the addresses and size of memory to be set will be rounded to refer to whole pages.

Setting **vm_inherit()** to VM_INHERIT_SHARE and forking a child task is the only way two Mach tasks can share physical memory. However, all the threads of a given task share all the same memory.

EXAMPLE

```
if ((ret = vm_allocate(task_self(), (vm_address_t *)&lock, sizeof(int),
        TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate returned value of ", ret);
    printf("Exiting with error.\n");
    exit(-1);
}
if ((ret = vm_inherit(task_self(), (vm_address_t)lock, sizeof(int),
        VM_INHERIT_SHARE)) != KERN_SUCCESS) {
    mach_error("vm_inherit returned value of ", ret);
    printf("Exiting with error.\n");
    exit(-1);
}
```

RETURN KERN_SUCCESS: The inheritance has been set.

KERN_INVALID_ADDRESS: Illegal address specified.

SEE ALSO **task_create()**, **vm_region()**

vm_protect()

SUMMARY Protect virtual memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_protect(**vm_task_t *target_task*, vm_address_t *address*, vm_size_t€*size*, boolean_t *set_maximum*, vm_prot_t *new_protection***)**

ARGUMENTS *target_task*: Task whose virtual memory is to be affected.

address: Starting address (this gets rounded down to a page boundary).

size: Size in bytes of the region for which protection is to change (this gets rounded up to a page boundary).

set_maximum: If set, make the protection change apply to the maximum protection associated with this address range; otherwise, change the current protection on this range. If the maximum protection is reduced below the current protection, both will be changed to reflect the new maximum.

new_protection: A new protection value for this region; either VM_PROT_NONE or some combination of VM_PROT_READ, VM_PROT_WRITE, and VM_PROT_EXECUTE.

DESCRIPTION The function **vm_protect()** changes the protection of some pages of allocated memory in a task's address space. In general, a protection value permits the named operation. When memory is first allocated it has all protection bits on. The exact interpretation of a protection value is machine-dependent. In the NeXT Mach operating system, three levels of memory protection are provided:

· No access
· Read and execute access
· Read, execute, and write access

VM_PROT_NONE permits no access. VM_PROT_WRITE permits read, execute, and write access; VM_PROT_READ or VM_PROT_EXECUTE permits read and execute access, but not write access.

EXAMPLE
```
vm_address_t    addr = (vm_address_t)mlock;

r = vm_protect(task_self(), addr, vm_page_size, FALSE, 0);
if (r != KERN_SUCCESS) {
    mach_error("vm_protect 0", r);
    exit(1);
}
printf("protect on\n");
```

RETURN KERN_SUCCESS: The memory has been protected.

KERN_PROTECTION_FAILURE: An attempt was made to increase the current or maximum protection beyond the existing maximum protection value.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified.

vm_read()

SUMMARY Read virtual memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_read(**vm_task_t *target_task*, vm_address_t *address*, vm_size_t *size*, pointer_t **data*, unsigned int **data_count***)**

ARGUMENTS *target_task*: Task whose memory is to be read.

address: The first address to be read (must be on a page boundary).

size: The number of bytes of data to be read (must be a multiple of **vm_page_size**).

data: The array of data copied from the given task.

data_count: Returns the size of the data array in bytes (will be an integral number of€pages).

DESCRIPTION The function **vm_read()** allows one task's virtual memory to be read by another task. The€data array is returned in a newly allocated region; the task reading the data should call€**vm_deallocate()** on this region when it's done with the data.

For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics()**.

EXAMPLE

```
if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data1,
        vm_page_size, TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate returned value of ", rtn);
    printf("vmread: Exiting.\n");
    exit(-1);
}

temp = data1;
for (i = 0; (i < vm_page_size); i++)
    temp[i] = i;
printf("Filled space allocated with some data.\n");
printf("Doing vm_read....\n");
if ((rtn = vm_read(task_self(), (vm_address_t)data1, vm_page_size,
        (pointer_t *)&data2, &data_cnt)) != KERN_SUCCESS) {
    mach_error("vm_read returned value of ", rtn);
    printf("vmread: Exiting.\n");
    exit(-1);
}
printf("Successful vm_read.\n");
```

RETURN KERN_SUCCESS: The memory has been read.

KERN_INVALID_ARGUMENT: Either *address* does not start on a page boundary or *size* isn't an integral number of pages.

KERN_NO_SPACE: There isn't enough room in the caller's virtual memory to allocate space for the data to be returned.

KERN_PROTECTION_FAILURE: The address region in the target task is protected against reading.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified, or there were not *size* bytes of data following that address.

SEE ALSO **vm_write()**, **vm_copy()**, **vm_deallocate()**

vm_region()

SUMMARY Get information about virtual memory regions

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_region(**vm_task_t *target_task*, vm_address_t **address*, vm_size_t€**size*, vm_prot_t **protection*, vm_prot_t **max_protection*, vm_inherit_t€**inheritance*, boolean_t **shared*, port_t **object_name*, vm_offset_t€**offset***)**

ARGUMENTS *target_task*: The task for which an address space description is requested.

address: The address at which to start looking for a region. On return, *address* will contain the start of the region (therefore, the value returned will be different from the value that was passed in if the specified region is part of a larger region).

size: Returns the size (in bytes) of the located region.

protection: Returns the current protection of the region.

max_protection: Returns the maximum allowable protection for this region.

inheritance: Returns the inheritance attribute for this region.

shared: Returns true if this region is shared, false if it isn't.

object_name: Returns the port identifying the region's memory object.

offset: Returns the offset into the pager object at which this region begins.

DESCRIPTION The **vm_region()** function returns a description of the specified region of the target task's virtual address space. This function begins at *address*, looking forward through memory until it comes to an allocated region. (If *address* is in a region, that region is used.) If *address* isn't in a region, it's set to the start of the first region that follows the incoming value. In this way an entire address space can be scanned. You can set *address* to the constant VM_MIN_ADDRESS (defined in the header file **mach/machine/vm_param.h**) to specify the first address in the address space.

EXAMPLE

```
char           *data;
kern_return_t r;
vm_size_t     size;
vm_prot_t     protection, max_protection;
vm_inherit_t  inheritance;
boolean_t     shared;
port_t        object_name;
vm_offset_t   offset;
vm_address_t  address;

/* . . . */
/* Check the inheritance of "data". */
address = (vm_address_t)&data;
r = vm_region(task_self(), &address, &size, &protection,
    &max_protection, &inheritance, &shared, &object_name, &offset);

if (r != KERN_SUCCESS)
    mach_error("Error calling vm_region", r);
else {
    printf("Inheritance is:  ");
    switch (inheritance) {
        case VM_INHERIT_SHARE:
            printf("Share with child\n");
            break;
        case VM_INHERIT_COPY:
            printf("Copy into child\n");
            break;
        case VM_INHERIT_NONE:
            printf("Absent from child\n");
            break;
        case VM_INHERIT_DONATE_COPY:
            printf("Copy and delete\n");
            break;
    }
}
```

RETURN KERN_SUCCESS: The region was located and information has been returned.

KERN_NO_SPACE: The task contains no region at or above *address*.

SEE ALSO **vm_allocate()**, **vm_deallocate()**, **vm_protect()**, **vm_inherit()**

vm_set_policy()

SUMMARY Set the paging policy for a region of memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_set_policy(**vm_task_t *target_task*, vm_address_t *address*, vm_size_t€*size*, int€*policy***)**

ARGUMENTS *target_task*: Task whose virtual memory is to be affected.

address: Starting address (must be on a page boundary).

size: Number of bytes (must be a multiple of **vm_page_size**).

policy: Mask specifying the paging policy. Values for *policy* are defined in the header file **mach/vm_policy.h**.

DESCRIPTION This function lets you control the paging policy for a region of memory. In addition to its normal paging policy, the system can control the placement of pages under certain patterns of access. These patterns are currently limited to strictly sequential access in either direction.

The *policy* mask can have the following values, which can be combined:

- VM_POLICY_PAGE_AHEAD (page ahead)
- VM_POLICY_SEQ_DEACTIVATE (deactivate behind)
- VM_POLICY_SEQ_FREE (free behind)
- VM_POLICY_RANDOM (don't use a special paging policy)

Note: The page-ahead policy isn't currently implemented.

Calls to **vm_policy()** affect memory at the backing store level, not the mapping level. For example, calling **vm_policy()** on a memory-mapped file affects the underlying file, and consequently all uses of that file. It is currently impossible for different users of the same file to have different policies for that file.

SEE ALSO **vm_deactivate()**

vm_statistics()

SUMMARY Examine virtual memory statistics

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_statistics(**vm_task_t *target_task*, vm_statistics_data_t **vm_stats***)**

ARGUMENTS *target_task*: The task that's requesting the statistics.

vm_stats: Returns the statistics.

DESCRIPTION The function **vm_statistics()** returns statistics about the kernel's use of virtual memory since the kernel was booted. The system page size is contained in both the **pagesize** field of the *vm_status* and the global variable **vm_page_size**, which is set at task initialization and remains constant for the life of the task.

```
struct vm_statistics {
    long  pagesize;         /* page size in bytes */
    long  free_count;       /* number of pages free */
    long  active_count;     /* number of pages active */
    long  inactive_count;   /* number of pages inactive */
    long  wire_count;       /* number of pages wired down */
    long  zero_fill_count; /* number of zero-fill pages */
    long  reactivations;    /* number of pages reactivated */
    long  pageins;          /* number of pageins */
    long  pageouts;         /* number of pageouts */
    long  faults;           /* number of faults */
    long  cow_faults;       /* number of copy-on-writes */
    long  lookups;          /* object cache lookups */
    long  hits;             /* object cache hits */
};
typedef struct vm_statistics   vm_statistics_data_t;
```

EXAMPLE
```
result=vm_statistics(task_self(), &vm_stats);
if (result != KERN_SUCCESS)
    mach_error("An error calling vm_statistics()!", result);
else
    printf("%d bytes of RAM are free\n",
        vm_stats.free_count * vm_stats.pagesize);
```

RETURN KERN_SUCCESS: The operation was successful.

vm_write()

SUMMARY Write virtual memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_write(**vm_task_t *target_task*, vm_address_t *address*, pointer_t *data*, unsigned int *data_count***)**

ARGUMENTS *target_task*: Task whose memory is to be written.

address: Starting address in task to be affected (must be a page boundary).

data: An array of bytes to be written.

data_count: The size in bytes of the data array (must be a multiple of **vm_page_size**).

DESCRIPTION The function **vm_write()** allows a task's virtual memory to be written by another task. For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics()**.

RETURN KERN_SUCCESS: Memory written.

KERN_INVALID_ARGUMENT: The address doesn't start on a page boundary, or the size isn't an integral number of pages.

KERN_PROTECTION_FAILURE: The address region in the target task is protected against writing.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified or the amount of allocated memory starting at *address* was less than *data_count*.

SEE ALSO **vm_copy()**, **vm_protect()**, **vm_read()**, **vm_statistics()**

Bootstrap Server Functions

The Bootstrap Server, like the Network Name Server, lets tasks publish ports that other tasks can send messages to. Unlike the Network Name Server, the Bootstrap Server is designed so that each server and its clients must be on the same host. The Bootstrap Server accomplishes this by using each task's bootstrap port (which is inherited from its parent) to ensure that the task is a descendent of a local task.

When a task forks a child task that shouldn't have access to the same set of services as the parent, the parent task must change its own bootstrap portÐperhaps only temporarilyÐso that its child inherits a *subset port*. The parent should then change the set of services available on the subset port to suit the child's requirements.

The Bootstrap Server was created by NeXT, so these functions aren't in other versions of Mach. See **/NextDeveloper/Headers/servers/bootstrap.defs** for more information of how the Bootstrap Server works.

Note: If possible, you should use Distributed Objects instead of the Bootstrap Server. The Distributed Objects system is described in the *NEXTSTEP General Reference*.

bootstrap_check_in()

SUMMARY Get receive rights to a service port

SYNOPSIS **#import <mach/mach.h>**
#import <servers/bootstrap.h>

kern_return_t **bootstrap_check_in(**port_t *bootstrap_port*, name_t *service_name*, port_all_t **service_port***)**

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_name: The string that names the service.

service_port: Returns receive rights to the service port.

DESCRIPTION Use this function in a server to start providing a service. The service must already be defined, either by the appropriate line in **/etc/bootstrap.conf** or by a call to **bootstrap_create_service()**. Calling **bootstrap_check_in()** makes the service active.

EXAMPLE

```
/* Get receive rights for our service. */
result=bootstrap_check_in(bootstrap_port, MYNAME, &my_service_port);
if (result != BOOTSTRAP_SUCCESS)
    mach_error("Couldn't create service", result);
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

BOOTSTRAP_UNKNOWN_SERVICE: The service doesn't exist. It might be defined in a subset (see **bootstrap_subset()**).

BOOTSTRAP_SERVICE_ACTIVE: The service has already been registered or checked in and the server hasn't died.

Returns appropriate kernel errors on RPC failure.

bootstrap_create_service()

SUMMARY Create a service and service port

SYNOPSIS **#import <mach/mach.h>**
#import <servers/bootstrap.h>

kern_return_t **bootstrap_create_service(**port_t *bootstrap_port*, name_t *service_name*, port_t **service_port***)**

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_name: The string that specifies the service.

service_port: Returns send rights for the service.

DESCRIPTION Creates a service named *service_name* and returns send rights to that port in *service_port*. The port may later be checked in as if this port were configured in the bootstrap configuration file. (At that time **bootstrap_check_in()** will return receive rights to *service_port* and will make the service active.)

This function is often used to create services that are available only to a subset of tasks (see **bootstrap_subset()**). Any task can call this functionÐit doesn't have to be the server.

EXAMPLE
```
/* Tell the bootstrap server about a service. */
result=bootstrap_create_service(bootstrap_port, SERVICENAME,
    &service_port);
if (result!=BOOTSTRAP_SUCCESS)
    mach_error("Couldn't create service", result);
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

BOOTSTRAP_SERVICE_ACTIVE: The service already exists.

Returns appropriate kernel errors on RPC failure.

bootstrap_info()

SUMMARY Get information about all known services

SYNOPSIS **#import <mach/mach.h>**
#import <servers/bootstrap.h>

kern_return_t **bootstrap_info(**port_t *bootstrap_port*, name_array_t **service_names*, unsigned int **service_names_count*, name_array_t **server_names*, unsigned€int€**server_names_count*, bool_array_t **service_active*, unsigned€int€**service_active_count***)**

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_names: Returns the names of all known services.

service_names_count: Returns the number of service names.

server_names: Returns the name, if known, of the server that provides the corresponding service. Except for the **mach_init** server, this name isn't known unless the bootstrap configuration file has a **server** line for this server.

server_names_count: Returns the number of server names.

service_active: Returns an array of booleans that correspond to the *service_names* array. For each item, the boolean value is true if the service is receiving messages sent to its port; otherwise, it's false.

service_active_count: Returns the number of items in the *service_active* array.

DESCRIPTION This function returns information about all services that are known. Note that it won't return information on services that are defined only in subsets, unless the subset port is an ancestor of *bootstrap_port*. (See **bootstrap_subset()** for information on subsets.)

EXAMPLE
```
result = bootstrap_info(bootstrap_port, &service_names, &service_cnt,
  &server_names, &server_cnt, &service_active, &service_active_cnt);
if (result != BOOTSTRAP_SUCCESS)
    printf("ERROR:  info failed: %d", result);
else {
    for (i = 0; i < service_cnt; i++)
        printf("Name: %-15s   Server: %-15s    Active: %-4s",
            service_names[i],
            server_names[i][0] == '\0' ? "Unknown" : server_names[i],
            service_active[i] ? "Yes\n" : "No\n");
}
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NO_MEMORY: The Bootstrap Server couldn't allocate enough memory to return the information.

Returns appropriate kernel errors on RPC failure.

bootstrap_look_up()

SUMMARY Get the service port of a particular service

SYNOPSIS **#import <mach/mach.h>**
#import <servers/bootstrap.h>

kern_return_t **bootstrap_look_up(**port_t *bootstrap_port*, name_t *service_name*, port_t€**service_port***)**

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_name: The string that identifies the service.

service_port: Returns send rights for the service port.

DESCRIPTION Returns send rights for the service port of the specified service. The service isn't guaranteed to be active. (To check whether the service is active, use **bootstrap_status()**.)

EXAMPLE
```
result=bootstrap_look_up(bootstrap_port, "FreeService2", &srvc_port);
if (result!=BOOTSTRAP_SUCCESS)
    printf("lookup failed: %d\n", result);
else {
    /* Access the service by sending messages to srvc_port. */
}
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_UNKNOWN_SERVICE: The service doesn't exist. It might be defined in a subset (see **bootstrap_subset()**).

Returns appropriate kernel errors on RPC failure.

bootstrap_look_up_array()

SUMMARY Get the service ports for an array of services

SYNOPSIS **#import <mach/mach.h>**
#import <servers/bootstrap.h>

kern_return_t **bootstrap_look_up_array(**port_t *bootstrap_port*, name_array_t€*service_names*, unsigned int *service_names_count*, port_array_t€**service_ports*, unsigned int **service_ports_count*, boolean_t€**all_services_known***)**

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_names: An array of service names.

service_names_count: The number of service names.

service_ports: Returns an array of service ports.

service_ports_count: Returns the number of service ports. This should be equal to *service_names_count*.

all_services_known: Returns true if every service name was recognized; otherwise returns€false.

DESCRIPTION Returns port send rights in corresponding entries of the array *service_ports* for all services named in the array *service_names*. You should call **vm_deallocate()** on *service_ports* when you no longer need it.

Unknown service names have the corresponding service port set to PORT_NULL. Note that these services might be available in a subset (see **bootstrap_subset()**).

EXAMPLE

```
kern_return_t   result;
port_t          my_bootstrap_port;
unsigned int    port_cnt;
boolean_t       all_known;
name_t          name_array[2]={"Service", "NetMessage"};
port_array_t    ports;

result = task_get_bootstrap_port(task_self(), &my_bootstrap_port);
if (result != KERN_SUCCESS) {
    mach_error("Couldn't get bootstrap port", result);
    exit(1);
}

result=bootstrap_look_up_array(my_bootstrap_port, name_array, 2,
    &ports, &port_cnt, &all_known);
if (result!=BOOTSTRAP_SUCCESS)
    mach_error("Lookup array failed", result);
else
    printf("Port count = %d, all known = %d\n", port_cnt, all_known);

/* . . . */
result=vm_deallocate(task_self(), (vm_address_t)ports,
    sizeof(ports)*port_cnt);
if (result != KERN_SUCCESS)
    mach_error("Trouble freeing ports", result);
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_BAD_COUNT: *service_names_count* was too large (greater than BOOTSTRAP_MAX_LOOKUP_COUNT, which is defined in the header file **server/bootstrap_defs.h**).

Returns appropriate kernel errors on RPC failure.

bootstrap_register()

SUMMARY Register send rights for a service port

SYNOPSIS **#import <mach/mach.h>**
#import <servers/bootstrap.h>

kern_return_t **bootstrap_register(**port_t *bootstrap_port*, name_t *service_name*, port_t€*service_port***)**

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_name: The string that identifies the service.

service_port: The service port for the service.

DESCRIPTION You can use this function to create a server that hasn't been defined in the bootstrap configuration file. This function specifies to the Bootstrap Server exactly which port should be the service port.

You can't register a service if an active binding already exists. However, you can register a service if the existing binding is inactive (that is, the Bootstrap Server currently holds receive rights for the service port); in this case the previous service port will be deallocated.

A service that is restarting can resume service for previous clients by setting *service_port* to the previous service port. You can get this port by calling **bootstrap_check_in()**.

EXAMPLE

```
/* Create a port to use as the service port. */
result=port_allocate(task_self(), &myport);
if (result != KERN_SUCCESS) {
    mach_error("Couldn't allocate a service port", result);
    exit(1);
}

/* Tell the bootstrap server about my service. */
result=bootstrap_register(bootstrap_port, MYNAME, myport);
if (result != BOOTSTRAP_SUCCESS)
    printf("Call to bootstrap_register failed: %d", result);
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

BOOTSTRAP_NAME_IN_USE: The service is already active.

Returns appropriate kernel errors on RPC failure.

bootstrap_status()

SUMMARY Check whether a service is available

SYNOPSIS **#import <mach/mach.h>**
#import <servers/bootstrap.h>

kern_return_t **bootstrap_status(**port_t *bootstrap_port*, name_t *service_name*, boolean_t€**service_active*????**)**

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_name: The string that specifies a particular service.

service_active: Returns true if the service is active; otherwise, returns false.

DESCRIPTION This function tells you whether a service is known to users of *bootstrap_port*, and whether it's active. A service is active if a server is able to receive messages on its service port. If a service isn't active, the Bootstrap Server holds receive rights for the service port.

EXAMPLE

```
result=bootstrap_status(bootstrap_port, MYNAME, &service_active);
if (result!=BOOTSTRAP_SUCCESS)
    printf("status check failed\n");
else {
    if (service_active)
        printf("Server %s is active\n", MYNAME);
    else
        printf ("Server %s is NOT active\n", MYNAME);
}
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_UNKNOWN_SERVICE: The service doesn't exist. It might be defined in a subset (see

bootstrap_subset()).

Returns appropriate kernel errors on RPC failure.

bootstrap_subset()

SUMMARY Get a new port to use as a bootstrap port

SYNOPSIS **#import <mach/mach.h>**
#import <servers/bootstrap.h>

kern_return_t **bootstrap_subset(**port_t *bootstrap_port*, port_t *requestor_port*, port_t€**subset_port***)**

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

requestor_port: A port that determines the life span of the subset.

subset_port: Returns the subset port.

DESCRIPTION Returns a new port to use as a bootstrap port. This port behaves exactly like the previous€*bootstrap_port*, with one exception: When you register a port by calling **bootstrap_register()** using *subset_port* as the bootstrap port, the registered port is available only to users of *subset_port* and its descendants. Lookups on the *subset_port* will€return ports registered specifically with this port, and will also return ports registered with ancestors of this *subset_port*. (The ancestors of *subset_port* are *bootstrap_port* and, if *bootstrap_port* is itself a subset port, any ancestors of *bootstrap_port*.)

You can override a service already registered with an ancestor port by registering it with the subset port. Any thread that looks up the service using the subset port will see only the version of the service that's registered with the subset port. This is one way to transparently provide services such as monitor programs or individualized spelling checkers, while the rest of the system still uses the default service.

When it's detected that *requestor_port* is destroyed, the subset port and its descendants are destroyed; the services advertised by these ports are destroyed, as well.

```
/* Get and save the current bootstrap port for this task. */
r = task_get_bootstrap_port(task_self(), &old_bs_port);
if (r != KERN_SUCCESS) {
    mach_error("task_get_bootstrap_port", r);
    exit(1);
}
/* Get a subset port. */
r = bootstrap_subset(old_bs_port, task_self(), &subset_port);
if (r != BOOTSTRAP_SUCCESS) {
    mach_error("Couldn't get unpriv port", r);
    exit(1);
}

/* Set the bootstrap port */
r = task_set_bootstrap_port(task_self(), subset_port);
if (r != KERN_SUCCESS) {
    mach_error("task_set_bootstrap_port", r);
    exit(1);
}
bootstrap_port = subset_port;
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

Returns appropriate kernel errors on RPC failure.

Network Name Server Functions

If possible, you should use Distributed Objects instead of the Network Name Server€functions.   The Distributed Objects system is described in the *NEXTSTEP General€Reference*.

netname_check_in()

SUMMARY Check a name into the local name space

SYNOPSIS **#import <mach/mach.h>**
#import <servers/netname.h>

kern_return_t **netname_check_in(**port_t *server_port*, netname_name_t *port_name*, port_t€*signature*, port_t *port_id***)**

ARGUMENTS *server_port*:   The task's port to the Network Name Server.   To use the system Network Name Server, this should be set to the global variable **name_server_port**.

port_name: The name of the port to be checked in.

signature: The port used to protect the right to remove a name.

port_id: The port to be checked in.

DESCRIPTION The function **netname_check_in()** enters a port with the name *port_name* into the name space of the local network server.   The *signature* argument is a port that's used to protect this name.   This same port must be presented on a **netname_check_out()** call for that call to be able to remove the name from the name space.

RETURN NETNAME_SUCCESS: The operation succeeded.

SEE ALSO **netname_check_out()**, **netname_look_up()**

netname_check_out()

SUMMARY Remove a name from the local name space

SYNOPSIS **#import <mach/mach.h>**
#import <servers/netname.h>

kern_return_t **netname_check_out(**port_t *server_port*, netname_name_t *port_name*, port_t *signature***)**

ARGUMENTS *server_port*:   The task's port to the Network Name Server.   To use the system Network Name Server, this should be set to **name_server_port**.

port_name: The name of the port to be checked out.

signature: The port used to protect the right to remove a name.

DESCRIPTION The function **netname_check_out()** removes a port with the name *port_name* from the name space of the local network server.   The *signature* argument must be the same port as the signature port passed to **netname_check_in()** when this name was checked in.

RETURN NETNAME_SUCCESS: The operation succeeded.

NETNAME_NOT_YOURS: The signature given to **netname_check_out()** did not match the signature with which the port was checked in.

SEE ALSO **netname_check_in()**, **netname_look_up()**

netname_look_up()

SUMMARY Look up a name on a specific host

SYNOPSIS **#import <mach/mach.h>**
#import <servers/netname.h>

kern_return_t **netname_look_up(**port_t *server_port*, netname_name_t *host_name*, netname_name_t *port_name*, port_t **port_id***)**

ARGUMENTS *server_port*: The task's port to the Network Name Server. To use the system Network Name Server, this should be set to **name_server_port**.

host_name: The name of the host to query. This can't be a null pointer.

port_name: The name of port to be looked up.

port_id: The port that was looked up.

DESCRIPTION The function **netname_look_up()** returns the value of the port named by *port_name* by questioning the host named by the *host_name* argument. Thus this call is a directed name lookup. The *host_name* may be any of the host's official nicknames. If it's an empty string, the local host is assumed. If *host_name* is ª*º, a broadcast lookup is performed.

Important: Use **NXPortNameLookup()** instead of **netname_look_up()** in all NEXTSTEP applications. (In the future, Listener instances might register with a server other than the Network Name Server.)

RETURN NETNAME_SUCCESS: The operation succeeded.

NETNAME_NOT_CHECKED_IN: **netname_look_up()** could not find the name at the given host.

NETNAME_NO_SUCH_HOST: The *host_name* argument to **netname_look_up()** does not name a valid host.

NETNAME_HOST_NOT_FOUND: **netname_look_up()** could not reach the host named by *host_name* (for instance, because it's down).

SEE ALSO **netname_check_in()**, **netname_check_out()**