CHAPTER 6 Thread Interface

This chapter discusses the specifics of the kernel's thread interfaces. This includes status functions related to threads. Properties associated with threads, such as special ports, are included here as well. Functions that apply to more than one thread appear in the task interface chapter.

Status:

Believed to be stable

catch_exception_raise

Server Interface — Handles the occurrence of an exception within a thread

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **catch_exception_raise**
(exception_port_t *exception_port,*
thread_port_t *thread,*
task_port_t *task,*
exception_type_t *exception,*
exception_data_t *code*,
mach_msg_type_number_t *code_count*);

catch_exception_raise_state

Expanded form

kern_return_t **catch_exception_raise_state**
(exception_port_t *exception_port,*
exception_type_t *exception,*
exception_data_t *code*,
mach_msg_type_number_t *code_count,*
thread_state_flavor_t* *flavor*,
thread_state_t *in_state*,
mach_msg_type_number_t *in_state_count*,
thread_state_t *out_state*,
mach_msg_type_number_t *out_state_count*);

catch_exception_raise_state_identity

Expanded form

kern_return_t **catch_exception_raise_state_identity**
(exception_port_t *exception_port,*
thread_port_t *thread*,
task_port_t *task*,
exception_type_t *exception,*
exception_data_t *code*,
mach_msg_type_number_t *code_count,*
thread_state_flavor_t* *flavor*,
thread_state_t *in_state*,
mach_msg_type_number_t *in_state_count*,
thread_state_t *out_state*,
mach_msg_type_number_t *out_state_count*);

DESCRIPTION

A **catch_exception_raise** function is called by **exc_server** as the result of a kernel message indicating that an exception occurred within a thread. *exception_port* is the port named via **thread_set_exception_ports** or **task_set_exception_ports** as the port that responds when the thread takes an exception.

The alternate message forms (the format being selected when the exception port was set) allow for selected thread state to be included.

PARAMETERS

exception_port
: [in exception (receive) right] The port to which the exception notification was sent.

thread
: [in thread-self send right] The thread self port for the thread taking the exception.

task
: [in task-self send right] The task self port for the task containing the thread taking the exception.

exception
: [in scalar] The type of the exception, as defined in **<mach/exception.h>**. The machine independent values raised by all implementations are:

EXC_BAD_ACCESS
: Could not access memory. *subcode* contains the bad memory address.

EXC_BAD_INSTRUCTION
: Instruction failed. Illegal or undefined instruction or operand.

EXC_ARITHMETIC
: Arithmetic exception; exact nature of exception is in *subcode* field.

EXC_EMULATION
: Emulation instruction. Emulation support instruction encountered. Details in *subcode* field.

EXC_SOFTWARE
: Software generated exception; exact exception is in *subcode* field. Codes 0 - 0xFFFF reserved to hardware; codes 0x10000 - 0x1FFFF reserved for OS emulation.

EXC_BREAKPOINT
: Trace, breakpoint, etc. Details in *subcode* field.

EXC_SYSCALL
: System call requested. Details in *subcode* field.

EXC_MACH_SYSCALL
: System call with a number in the Mach call range requested. Details in *subcode* field.

code
: [in scalar] A machine dependent array indicating a particular instance of *exception*.

code_count
: [in scalar] The size of the buffer (in natural-sized units).

flavor
: [pointer to in/out scalar] On input, the type of state included as selected when the exception port was set. On output, the type of state being returned.

in_state
: [pointer to in structure] State information of the thread at the time of the exception.

in_state_count
: [in scalar] The size of the buffer (in natural-sized units).

out_state
: [out structure] The state the thread will have if continued from the point of the exception. The maximum size of this array is THREAD_STATE_MAX.

out_state_count
: [out scalar] The size of the buffer (in natural-sized units).

NOTES

When an exception occurs in a thread, the thread sends an exception message to its exception port, blocking in the kernel waiting for the receipt of a reply. It is assumed that some task is listening (most likely with **mach_msg_server**) to this port, using the **exc_server** function to decode the messages and then call the linked in **catch_exception_raise**. It is the job of **catch_exception_raise** to handle the exception and decide the course of action for *thread*.

If the thread should continue from the point of exception, **catch_exception_raise** would return KERN_SUCCESS. This causes a reply message to be sent to the kernel, which will allow the thread to continue from the point of the exception.

If some other action should be taken by *thread*, the following actions should be performed by **catch_exception_raise**:

- **thread_suspend**. This keeps the thread from proceeding after the next step.
- **thread_abort**. This aborts the message receive operation currently blocking the thread.
- **thread_set_state** (if using the **catch_exception_raise** form). Set the thread's state so that it continues doing something else.
- **thread_resume**. Let the thread start running from its new state.
- Return a value other than KERN_SUCCESS so that no reply message is sent. (Actually, the kernel uses a send once right to send the exception message, which **thread_abort** destroys, so replying to the message is harmless.)

The thread can always be destroyed with **thread_terminate**.

A thread can have two exception ports active for it: its thread type specific exception port and the task type specific exception port. The kernel will try sending an exception message to both ports looking for a reply message with a return value of KERN_SUCCESS. The kernel tries the thread specific port first, then the task specific port. If the return value from the first exception message the kernel sends has a return value of KERN_SUCCESS, the thread continues (with a possibly modified state). If the return value is not KERN_SUCCESS, the kernel tries the second port. If that return value is KERN_SUCCESS, the thread continues; otherwise, the thread is terminated. To get the effect of a non-success return value, the server interface should return MIG_DESTROY_REQUEST. This causes **exc_server** and **mach_msg_server** to destroy the kernel's request (as opposed to sending a reply with a KERN_SUCCESS value).

RETURN VALUE

A return value of KERN_SUCCESS indicates that the thread is to continue from the point of exception. A return value of MIG_NO_REPLY indicates that the exception was handled directly and the thread was restarted or terminated by the exception handler. A return value of MIG_DESTROY_REQUEST causes the kernel to try another exception handler (or terminate the thread). Any other value will cause **mach_msg_server** to remove the task and thread port references.

RELATED INFORMATION

Functions: **exc_server**, **thread_abort**, **task_get_exception_ports**, **thread_get_exception_ports**, **thread_get_state**, **thread_resume**, **task_set_exception_ports, thread_set_exception_ports**, **task_swap_exception_ports, thread_swap_exception_ports**, **thread_set_state**, **thread_suspend**, **thread_terminate.**

mach_thread_self

Status:

Believed to be stable

System Trap — Returns the thread self port

LIBRARY

#include <**mach/mach_traps.h**>

SYNOPSIS

```
thread_port_t mach_thread_self
                ();
```

DESCRIPTION

The **mach_thread_self** function returns send rights to the thread's own kernel port.

PARAMETERS

None

RETURN VALUE

[thread-self send right] Send rights to the thread's port.

RELATED INFORMATION

Functions: **thread_info**, **thread_set_special_port**.

receive_samples

Status:

Experimental support for the profil system call

Server Interface — Handles the occurrence of a PC sampling message

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **receive_samples**

(sample_port_t	*sample_port,*
sample_array_t	*samples*,
mach_msg_type_number_t	*samplesCnt*);

DESCRIPTION

A **receive_samples** function is called by **prof_server** as the result of a kernel message indicating that a set of program counter samples has been gathered. *sample_port* is the port named via **task_sample** or **thread_sample**.

PARAMETERS

sample_port
: [in sample (receive) right] The port to which the sample message was sent.

sample
: [pointer to in array of *vm_address_t*] An array of PC sample values.

sampleCnt
: [in scalar] The number of values in *sample*.

NOTES

This interface is machine word length specific because of the virtual addresses in the *samples* parameter.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **task_sample**, **thread_sample**, **prof_server.**

Status:

Believed to be stable

thread_abort

Function — Aborts a thread

SYNOPSIS

kern_return_t **thread_abort**
(thread_port_t *target_thread)*;

DESCRIPTION

The **thread_abort** function aborts page faults and any message primitive calls in use by *target_thread*. Scheduling depressions and clock sleeps are also aborted. The call returns a code indicating that it was interrupted. The call is interrupted even if the thread (or the task containing it) is suspended. If it is suspended, the thread receives the interrupt when it resumes.

If its state is not modified before it resumes, the thread will retry an aborted page fault. The Mach message trap returns either MACH_SEND_INTERRUPTED or MACH_RCV_INTERRUPTED, depending on whether the send or the receive side was interrupted. Note, though, that the Mach message trap is contained within the **mach_msg** library routine, which, by default, retries interrupted message calls.

The basic purpose of **thread_abort** is to let one thread cleanly stop another thread (*target_thread*). The target thread is stopped in such a manner that its future execution can be controlled in a predictable way. When **thread_abort** returns, the target thread will appear to have just returned from the kernel (if it had been in kernel mode).

PARAMETERS

target_thread
[in thread send right] The thread to be aborted.

NOTES

By way of comparison, 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** function returns the thread's user state, while **thread_set_state** allows modification of the user state.

A problem occurs if a suspended thread had been executing within a system call. In this case, the thread has, not only a user state, but an associated kernel state. (The kernel state cannot be changed with **thread_set_state**.) As a result, when the thread resumes, the system call can return, producing a change in the user state and, possibly, user memory.

For a thread executing within a system call, **thread_abort** aborts the kernel call from the thread's point of view. Specifically, it resets the kernel state so that the thread will resume execution at the system call return, with the return code value set to one of the interrupted codes. The system call itself may be completed entirely, aborted entirely or be partially completed, depending on when the abort is received. As a result, if the thread's user state has been modified by **thread_set_state**, it will not be altered un-predictably by any unexpected system call side effects.

For example, to simulate a POSIX signal, use the following sequence of calls:

- **thread_suspend** — To stop the thread.
- **thread_abort** — To interrupt any system call in progress and set the return value to "interrupted". Because the thread is already stopped, it will not return to user code.
- **thread_set_state** — To modify the thread's user state to simulate a procedure call to the signal handler.
- **thread_resume** — To resume execution at the signal handler. If the thread's stack is set up correctly, the thread can return to the interrupted system call. Note that the code to push an extra stack frame and change the registers is highly machine dependent.

CAUTIONS

As a rule, do not use **thread_abort** on a non-suspended thread. This operation is very risky because it is difficult to know which system trap, if any, is executing and whether an interrupt return will result in some useful action by the thread.

thread_abort will abort any non-atomic operation (such as a multi-page **memory_object_data_supply**) at an arbitrary point in a non-restartable way. Such problems can be avoided by using **thread_abort_safely**.

RETURN VALUE

KERN_EXCEPTION_PROTECTED
: The thread is processing a protected exception.

RELATED INFORMATION

Functions: **mach_msg**, **thread_get_state**, **thread_info**, **thread_set_state**, **thread_suspend**, **thread_terminate**, **thread_abort_safely**, **thread_set_exception_ports**.

Status:

Believed to be stable

thread_abort_safely

Function — Aborts a thread, restartably

SYNOPSIS

```
kern_return_t thread_abort_safely
        (thread_port_t                    target_thread);
```

DESCRIPTION

The **thread_abort_safely** function aborts page faults and any message primitive calls in use by *target_thread*. Scheduling depressions and clock sleeps are also aborted. The call returns a code indicating that it was interrupted. The call is interrupted even if the thread (or the task containing it) is suspended. If it is suspended, the thread receives the interrupt when it resumes.

If its state is not modified before it resumes, the thread will retry an aborted page fault. The Mach message trap returns either MACH_SEND_INTERRUPTED or MACH_RCV_INTERRUPTED, depending on whether the send or the receive side was interrupted. Note, though, that the Mach message trap is contained within the **mach_msg** library routine, which, by default, retries interrupted message calls.

The basic purpose of **thread_abort_safely** is to let one thread cleanly stop another thread (*target_thread*). The target thread is stopped in such a manner that its future execution can be controlled in a predictable way. When **thread_abort_safely** returns (if successful), the target thread will appear to have just returned from the kernel (if it had been in kernel mode).

PARAMETERS

target_thread
: [in thread send right] The thread to be aborted.

NOTES

By way of comparison, 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** function returns the thread's user state, while **thread_set_state** allows modification of the user state.

A problem occurs if a suspended thread had been executing within a system call. In this case, the thread has, not only a user state, but an associated kernel state. (The kernel state cannot be changed with **thread_set_state**.) As a result, when the thread resumes, the system call can return, producing a change in the user state and, possibly, user memory.

For a thread executing within a system call, **thread_abort_safely** aborts the kernel call from the thread's point of view. Specifically, it resets the kernel state so that the thread will resume execution at the system call return, with the return code value set to one of the interrupted codes. The system call itself may completed entirely, aborted entirely or be partially completed, depending on when the abort is received. As a result, if the thread's user state has been modified by **thread_set_state**, it will not be altered un-predictably by any unexpected system call side effects.

For example, to simulate a POSIX signal, use the following sequence of calls:

- **thread_suspend** — To stop the thread.
- **thread_abort_safely** — To interrupt any system call in progress and set the return value to "interrupted". Because the thread is already stopped, it will not return to user code.
- **thread_set_state** — To modify the thread's user state to simulate a procedure call to the signal handler.
- **thread_resume** — To resume execution at the signal handler. If the thread's stack is set up correctly, the thread can return to the interrupted system call. Note that the code to push an extra stack frame and change the registers is highly machine dependent.

CAUTIONS

As a rule, do not use **thread_abort_safely** on a non-suspended thread. This operation is very risky because it is difficult to know which system trap, if any, is executing and whether an interrupt return will result in some useful action by the thread.

thread_abort_safely will not abort any non-atomic operation (such as a multi-page **memory_object_data_supply** or exception processing) but will return an error instead. The caller of this function must then allow the thread to resume and attempt to abort it later. If the thread must be aborted, even if doing so would abort any non-atomic operations, **thread_abort** would be used.

RETURN VALUE

KERN_FAILURE
The thread is in the middle of a non-restartable operation.

RELATED INFORMATION

Functions: **mach_msg**, **thread_get_state**, **thread_info**, **thread_set_state**, **thread_suspend**, **thread_terminate**, **thread_abort**.

thread_activation_create

Status:

Experimental: [OSF-RI MK6]

Function — Create a thread activation

SYNOPSIS

kern_return_t **thread_activation_create**

(task_port_t	*task,*
mach_port_t	*RPC_port,*
vm_offset_t	*user_stack,*
mach_port_t*	*new_act*);

DESCRIPTION

The **thread_activation_create** function creates a thread activation, or "empty thread", into which a client thread shuttle can migrate during RPC. The *RPC_port* must name either a receive right held by the task, or a port set in the task. The new thread activation will be added to a pool of activations attached to this port (or port set). Incoming RPC's targeted at the port (or one of the ports in the set) can use any of the activations in the pool. That is, the client thread shuttle will migrate into the specified server task, take one of the thread activations out of the pool, and join up with it for the duration of the RPC. The shuttle will migrate back to the original client activation at the end of the RPC. If no thread activations are in the pool, RPC will be blocked until one is created in the pool, or an existing one finishes its RPC and returns to the pool.

The new thread activation will begin each RPC using the stack pointer specified by *user_stack*. The kernel neither knows nor cares how big the specified stack is.

When a short-circuited RPC (or **mach_rpc**) is invoked on a port created with **mach_port_allocate_subsystem**, the RPC will begin execution in the subsystem server at the work function address specified by the port and routine number and looked up in the associated subsystem data.

PARAMETERS

task
: [in task send right] The port for the task that is to contain the new thread activation.

RPC_port
: [in receive right] A receive right held by the task, or a port set in the task.

user_stack
: [in scalar] The virtual address in the task to be used as the starting address of the user-level stack.

new_act

[out thread send right] The kernel-assigned name for the new thread activation.

NOTES

The following calls targeted at a thread_act port may _not_ be called on an empty thread_act (and will return KERN_INVALID_ARGUMENT if they are called with one):

- **thread_abort**
- **thread_abort_safely**
- **thread_depress_abort**
- **thread_info**
- **thread_wire**

In addition, if **thread_switch** is called with an empty thread_act as its first argument, the argument will be ignored (i.e., the function will behave as if a zero-valued argument had been given).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **mach_port_allocate_subsystem**.

Status:

Believed to be stable

thread_create

Function — Creates a thread within a task

SYNOPSIS

```
kern_return_t thread_create
        (task_port_t            parent_task,
         thread_port_t*         child_thread);
```

DESCRIPTION

The **thread_create** function creates a new thread within *parent_task*. The new thread has a suspend count of one and no processor state.

The new thread holds a send right for its thread kernel port. A send right for the thread's kernel port is also returned to the calling task or thread in *child_thread*. The new thread's exception ports are set to MACH_PORT_NULL.

PARAMETERS

parent_task
: [in task send right] The port for the task that is to contain the new thread.

child_thread
: [out thread send right] The kernel-assigned name for the new thread.

NOTES

To get a new thread running, first use **thread_set_state** to set a processor state for the thread. Then, use **thread_resume** to schedule the thread for execution. Alternately, use **thread_create_running**.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_threads**, **thread_get_special_port**, **thread_get_state**, **thread_resume**, **thread_set_special_port**, **thread_set_state**, **thread_suspend**, **thread_terminate**, **thread_create_running**.

thread_create_running

Status:

Believed to be stable

Function — Optimized creation of a running thread

SYNOPSIS

```
kern_return_t thread_create_running
        (task_port_t              parent_task,
         thread_state_flavor_t    flavor,
         thread_state_t           state,
         thread_port_t*           child_thread);
```

DESCRIPTION

The **thread_create_running** function creates a new thread within *parent_task*. The new thread has is not suspended. Its initial state is given by *state*. *flavor* specifies the type of state to set.

The format of the state to set is machine specific; it is defined in **<mach/thread_status.h>**.

The new thread holds a send right for its thread kernel port. A send right for the thread's kernel port is also returned to the calling task or thread in *child_thread*. The new thread's exception ports are set to MACH_PORT_NULL.

PARAMETERS

parent_task
: [in task send right] The port for the task that is to contain the new thread.

flavor
: [in scalar] The type of state to establish. Valid values correspond to supported machine architecture features.

state
: [pointer to in structure] State information for the specified thread.

child_thread
: [out thread send right] The kernel-assigned name for the new thread.

NOTES

This is an optimized form of the sequence: **thread_create**, **thread_set_state** and **thread_resume**.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_threads**, **thread_get_special_port**, **thread_get_-state**, **thread_resume**, **thread_set_special_port**, **thread_set_state**, **thread_-suspend**, **thread_terminate**, **thread_create**.

thread_depress_abort

Status: **Believed to be stable**

Function — Cancel thread scheduling depression

SYNOPSIS

kern_return_t **thread_depress_abort**
(thread_port_t *thread*);

DESCRIPTION

The **thread_depress_abort** function cancels any scheduling depression effective for *thread* caused by a **thread_switch** call.

PARAMETERS

thread
[in thread send right] Thread whose scheduling depression is canceled.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **thread_abort**, **thread_switch**.

Status:

Believed to be stable

thread_get_exception_ports

Function — Returns a send right to an exception port

SYNOPSIS

kern_return_t **thread_get_exception_ports**

(thread_port_t	*thread,*
exception_mask_t	*exception_types*,
exception_mask_array_t	*old_exception_masks*,
mach_msg_type_number_t*	*old_exception_count*,
exception_port_array_t	*old_exception_ports*,
exception_behavior_array_t	*old_behaviors*,
thread_state_flavor_array_t	*old_flavors*);

DESCRIPTION

The **thread_get_exception_ports** function returns send rights for a specified set of exception ports belonging to *thread*. The call returns a set of quadruples <exception type mask, exception port, behavior, flavor> for each unique set of <exception port, behavior, flavor> in effect for the thread where the exception type mask indicates for which exception types the other values apply.

PARAMETERS

thread
: [in thread send right] The thread for which to return the exception ports.

exception_types
: [in scalar] A flag word indicating the types of exceptions for which the exception ports are desired.

EXC_MASK_BAD_ACCESS
: Could not access memory.

EXC_MASK_BAD_INSTRUCTION
: Instruction failed. Illegal or undefined instruction or operand.

EXC_MASK_ARITHMETIC
: Arithmetic exception.

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
Trace, breakpoint, etc.

EXC_MASK_SYSCALL
System call requested.

EXC_MASK_MACH_SYSCALL
System call with a number in the Mach call range requested.

old_exception_masks
[out array of *exception_mask_t*] An array, each element being a mask specifying for which exception types the corresponding element of the other arrays apply.

old_exception_count
[pointer to in/out scalar] On input, the maximum size of the array buffers; on output, the number of returned <exception type mask, exception port, behavior, flavor> sets returned.

old_exception_ports
[out array of exception send rights] The returned exception ports.

old_behaviors
[out array of *exception_behavior_t*] The type of exception message to be sent. Defined types are:

EXCEPTION_DEFAULT
Send a **catch_exception_raise** message including the thread identity.

EXCEPTION_STATE
Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_IDENTITY
Send a **catch_exception_raise_state_identity** message including the thread identity and state.

old_flavors
[out array of *thread_state_flavor_t*] The type of state to be sent with the exception message. These types are defined in **<mach/thread_states.h>**.

RETURN VALUE

KERN_EXCEPTION_PROTECTED
One of the requested exception ports is protected and cannot be returned.

RELATED INFORMATION

Functions: **mach_thread_self**, **task_get_exception_ports**, **task_set_exception_ports**, **task_swap_exception_ports**, **thread_create**, **thread_set_exception_ports**, **thread_swap_exception_ports**, **catch_exception_raise**.

thread_get_special_port

Status:

Believed to be stable

Function — Returns a send right to a special port

SYNOPSIS

```
kern_return_t thread_get_special_port
                (thread_port_t                    thread,
                thread_special_port_t             which_port,
                mach_port_t*                      special_port);
```

thread_get_kernel_port

Macro form

```
kern_return_t thread_get_kernel_port
                (thread_port_t                    thread,
                mach_port_t*                      special_port)
```

⇒ **thread_get_special_port** (*thread*, THREAD_KERNEL_PORT, *special_port*)

DESCRIPTION

The **thread_get_special_port** function returns a send right for a special port belonging to *thread*.

The thread kernel port is a port for which the kernel holds the receive right. The kernel uses this port to identify the thread.

If one thread has a send right for the kernel port of another thread, it can use the port to perform kernel operations for the other thread. Send rights for a kernel port normally are held only by the thread to which the port belongs, or by the task that contains the thread. Using the **mach_msg** function, however, any thread can pass a send right for its kernel port to another thread.

PARAMETERS

thread
: [in thread send right] The thread for which to return the port's send right.

which_port
: [in scalar] The special port for which the send right is requested. Valid values are:

  THREAD_KERNEL_PORT
  : [thread-self send right] The port used to name the thread. Used to invoke operations that affect the thread. This is the port returned by **mach_thread_self**.

special_port
[out thread-special send right] The returned value for the port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_thread_self**, **task_get_special_port**, **task_set_special_port**, **thread_create**, **thread_set_special_port**.

thread_get_state

Status:

Believed to be stable

Function — Returns the execution state for a thread

SYNOPSIS

kern_return_t **thread_get_state**

(thread_port_t	*target_thread,*
thread_state_flavor_t	*flavor,*
thread_state_t	*old_state*,
mach_msg_type_number_t*	*old_state_count*);

DESCRIPTION

The **thread_get_state** function returns the execution state (for example, the machine registers) for *target_thread*. *flavor* specifies the type of state information returned.

The format of the data returned is machine specific; it is defined in <**mach/thread_status.h**>.

PARAMETERS

target_thread
: [in thread send right] The thread for which the execution state is to be returned. The calling thread cannot specify itself.

flavor
: [in scalar] The type of execution state to be returned. Valid values correspond to supported machined architectures.

old_state
: [out structure] State information for the specified thread.

old_state_count
: [in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_info**, **thread_info**, **thread_set_state**.

Status:

Believed to be stable

thread_info

Function — Returns information about a thread

SYNOPSIS

kern_return_t **thread_info**

(thread_port_t	*target_thread,*
thread_flavor_t	*flavor,*
thread_info_t	*thread_info*,
mach_msg_type_number_t*	*thread_info_count*);

DESCRIPTION

The **thread_info** function returns an information structure of type *flavor*.

PARAMETERS

target_thread
: [in thread send right] The thread for which the information is to be returned.

flavor
: [in scalar] The type of information to be returned. Valid values are:

THREAD_BASIC_INFO
: Returns basic information about the thread, such as the thread's run state and suspend count. The returned structure is **thread_basic_info**.

THREAD_SCHED_FIFO_INFO
: Returns FIFO scheduling policy information about the thread. The returned structure is **policy_fifo_info**

THREAD_SCHED_RR_INFO
: Returns round-robin scheduling policy information about the thread. The returned structure is **policy_rr_info**

THREAD_SCHED_TIMESHARE_INFO
: Returns timeshare scheduling policy information about the thread. The returned structure is **policy_timeshare_info**

thread_info
: [out structure] Information about the specified thread.

thread_info_count
: [in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

NOTES

At any given time, a thread has only one scheduling policy in effect for it. Thus, only one of the scheduling information structures will be valid, that so indicated by the policy value returned by THREAD_BASIC_INFO.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_info**, **task_threads**, **thread_get_state**, **thread_set_special_-port**, **thread_set_state**.

Data Structures: **thread_basic_info**, **policy_timeshare_info**, **policy_fifo_info**, **policy_rr_info**.

Status:

Believed to be stable

thread_resume

Function — Resumes a thread

SYNOPSIS

kern_return_t **thread_resume**
    (thread_port_t *target_thread)*;

DESCRIPTION

The **thread_resume** function decrements the suspend count for *target_thread* by one. The thread is resumed if its suspend count goes to zero. If the suspend count is still positive, **thread_resume** must be repeated until the count reaches zero.

PARAMETERS

target_thread
[in thread send right] The thread to be resumed.

NOTES

An attempt to lower the suspend count below zero is ignored.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_resume**, **task_suspend**, **thread_create**, **thread_info**, **thread_suspend**, **thread_terminate**.

thread_sample

Status:

Experimental support for the profil system call

Function — Perform periodic PC sampling for a thread

SYNOPSIS

kern_return_t **thread_sample**
(thread_port_t *sample_thread*,
sample_port_t *reply_port*);

DESCRIPTION

The **thread_sample** function causes the program counter (PC) of the specified *sample_thread* to be sampled periodically (whenever the thread happens to be running at the time of the kernel's "hardclock" interrupt). The set of PC sample values obtained are saved in buffers which are sent to the specified *reply_port* in **receive_samples** messages.

PARAMETERS

sample_thread
[in thread send right] Thread whose PC is to be sampled

reply_port
[in sample receive (to be converted to send) right] Port to which PC sample buffers are sent. A value of MACH_PORT_NULL stops PC sampling for the thread.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_sample**, **receive_samples**.

Status:

Believed to be stable

thread_set_exception_ports

Function — Sets exception ports for a thread

SYNOPSIS

kern_return_t **thread_set_exception_ports**

(thread_port_t	*thread,*
exception_mask_t	*exception_types,*
exception_port_t	*exception_port*,
exception_behavior_t	*behavior*,
thread_state_flavor_t	*flavor*);

DESCRIPTION

The **thread_set_exception_ports** function sets a specified set of exception ports belonging to *thread*.

PARAMETERS

thread
[in thread send right] The thread for which to set the ports.

exception_types
[in scalar] A flag word indicating the types of exceptions for which the exception port applies.

EXC_MASK_BAD_ACCESS
Could not access memory.

EXC_MASK_BAD_INSTRUCTION
Instruction failed. Illegal or undefined instruction or operand.

EXC_MASK_ARITHMETIC
Arithmetic exception.

EXC_MASK_EMULATION
Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
Software generated exception.

EXC_MASK_BREAKPOINT
Trace, breakpoint, etc.

EXC_MASK_SYSCALL
System call requested.

EXC_MASK_MACH_SYSCALL
System call with a number in the Mach call range requested.

exception_port
[in exception send right] The exception port for all selected exception types.

behavior
[in scalar] The type of exception message to be sent. Defined types are:

EXCEPTION_DEFAULT
Send a **catch_exception_raise** message including the thread identity.

EXCEPTION_DEFAULT_PROTECTED
Send a **catch_exception_raise** message including the thread identity. Mark the exception port (and associated exceptions) as protected.

EXCEPTION_STATE
Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_PROTECTED
Send a **catch_exception_raise_state** message including the thread state. Mark the exception port (and associated exceptions) as protected.

EXCEPTION_STATE_IDENTITY
Send a **catch_exception_raise_state_identity** message including the thread identity and state.

EXCEPTION_STATE_IDENTITY_PROTECTED
Send a **catch_exception_raise_state_identity** message including the thread identity and state. Mark the exception port (and associated exceptions) as protected.

flavor
[in scalar] The type of state to be sent with the exception message. These types are defined in **<mach/thread_states.h>**.

NOTES

If the value of the EXC_MACH_SYSCALL exception class exception port is the host name port, Mach kernel traps are executed by the kernel as expected; any other value causes the attempted execution of these system call numbers to be considered an exception.

A “protected” exception port is one which cannot be fetched and for which exception processing cannot be aborted (**thread_abort**).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_thread_self**, **task_get_exception_ports**, **task_set_exception_ports**, **task_swap_exception_ports**, **thread_create**, **thread_get_exception_ports**, **thread_swap_exception_ports**, **catch_exception_raise**, **thread_abort**.

thread_set_special_port

Status:

Believed to be stable

Function — Sets a special port for a thread

SYNOPSIS

kern_return_t **thread_set_special_port**
    (thread_port_t *thread,*
    thread_special_port_t *which_port,*
    mach_port_t *special_port*);

thread_set_kernel_port

Macro form

kern_return_t **thread_set_kernel_port**
    (thread_port_t *thread,*
    mach_port_t *special_port)*

⇒ **thread_set_special_port** (*thread*, THREAD_KERNEL_PORT, *special_port*)

DESCRIPTION

The **thread_set_special_port** function sets a special port belonging to *thread*.

PARAMETERS

thread
: [in thread send right] The thread for which to set the port.

which_port
: [in scalar] The special port to be set. Valid values are:

    THREAD_KERNEL_PORT
    : [thread-self port] The thread's kernel port. Used by the kernel to receive messages from the thread. This is the port returned by **mach_thread_self**.

special_port
: [in thread-special send right] The value for the port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_thread_self**, **task_get_special_port**, **task_set_special_port**, **thread_create**, **thread_get_special_port**.

Status:

Believed to be stable

thread_set_state

Function — Sets the execution state for a thread

SYNOPSIS

```
kern_return_t thread_set_state
        (thread_port_t                    target_thread,
         thread_state_flavor_t            flavor,
         thread_state_t                   new_state,
         mach_msg_type_number_t           new_state_count);
```

DESCRIPTION

The **thread_set_state** function sets the execution state (for example, the machine registers) for *target_thread*. *flavor* specifies the type of state to set.

The format of the state to set is machine specific; it is defined in **<mach/thread_status.h>**.

PARAMETERS

target_thread
: [in thread send right] The thread for which to set the execution state. The calling thread cannot specify itself.

flavor
: [in scalar] The type of state to set. Valid values correspond to supported machine architecture features.

new_state
: [pointer to in structure] State information for the specified thread.

new_state_count
: [in scalar] The size of the buffer (in natural-sized units).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_info**, **thread_get_state**, **thread_info**.

thread_suspend

Status:

Believed to be stable

Function — Suspends a thread

SYNOPSIS

kern_return_t **thread_suspend**
(thread_port_t *target_thread)*;

DESCRIPTION

The **thread_suspend** function increments the suspend count for *target_thread* and prevents the thread from executing any more user-level instructions.

In this context, a user-level instruction can be either a machine instruction executed in user mode or a system trap instruction, including a page fault. 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, the system trap returns when the thread resumes.

To resume a suspended thread, use **thread_resume**. If the suspend count is greater than one, **thread_resume** must be repeated that number of times.

PARAMETERS

target_thread
[in thread send right] The thread to be suspended.

CAUTIONS

Unpredictable results may occur if a program suspends a thread and alters its user state so that its direction is changed upon resuming. Note that the **thread_abort** function allows a system call to be aborted only if it is progressing in a predictable way.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_resume**, **task_suspend**, **thread_abort**, **thread_get_state**, **thread_info**, **thread_resume**, **thread_set_state**, **thread_terminate**.

Status:

Believed to be stable

thread_swap_exception_ports

Function — Swaps exception ports for a thread

SYNOPSIS

kern_return_t **thread_swap_exception_ports**

(thread_port_t	*thread,*
exception_mask_t	*exception_types,*
exception_port_t	*exception_port*,
exception_behavior_t	*behavior*,
thread_state_flavor_t	*flavor*,
exception_mask_array_t	*old_exception_masks*,
mach_msg_type_number_t*	*old_exception_count*,
exception_port_array_t	*old_exception_ports*,
exception_behavior_array_t	*old_behaviors*,
thread_state_flavor_array_t	*old_flavors*);

DESCRIPTION

The **thread_swap_exception_ports** function sets a specified set of exception ports belonging to *thread*, returning the old set.

PARAMETERS

thread
: [in thread send right] The thread for which to set the ports.

exception_types
: [in scalar] A flag word indicating the types of exceptions for which the exception port applies.

EXC_MASK_BAD_ACCESS
: Could not access memory.

EXC_MASK_BAD_INSTRUCTION
: Instruction failed. Illegal or undefined instruction or operand.

EXC_MASK_ARITHMETIC
: Arithmetic exception

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
: Trace, breakpoint, etc.

EXC_MASK_SYSCALL
: System call requested.

EXC_MASK_MACH_SYSCALL
: System call with a number in the Mach call range requested.

exception_port
: [in exception send right] The exception port for all selected exception types.

behavior
: [in scalar] Control of the behavior of the exception processing. Defined types are:

EXCEPTION_DEFAULT
: Send a **catch_exception_raise** message including the thread identity.

EXCEPTION_DEFAULT_PROTECTED
: Send a **catch_exception_raise** message including the thread identity. Mark the exception port (and associated exceptions) as protected.

EXCEPTION_STATE
: Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_PROTECTED
: Send a **catch_exception_raise_state** message including the thread state. Mark the exception port (and associated exceptions) as protected.

EXCEPTION_STATE_IDENTITY
: Send a **catch_exception_raise_state_identity** message including the thread identity and state.

EXCEPTION_STATE_IDENTITY_PROTECTED
: Send a **catch_exception_raise_state_identity** message including the thread identity and state. Mark the exception port (and associated exceptions) as protected.

flavor
: [in scalar] The type of state to be sent with the exception message. These types are defined in **<mach/thread_states.h>**.

old_exception_masks
: [out array of *exception_mask_t*] An array, each element being a mask specifying for which exception types the corresponding element of the other arrays apply.

old_exception_count
: [pointer to in/out scalar] On input, the maximum size of the array buffers; on output, the number of returned <exception type mask, exception port, behavior, flavor> sets returned.

old_exception_ports
: [out array of exception send rights] The returned exception ports.

old_behaviors
: [out array of *exception_behavior_t*] The type of exception message to be sent as with *behavior*.

old_flavors
: [out array of *thread_state_flavor_t*] The type of state to be sent with the exception message. These types are defined in **<mach/thread_states.h>**.

NOTES

If the value of the EXC_MACH_SYSCALL exception class exception port is the host name port, Mach kernel traps are executed by the kernel as expected; any other value causes the attempted execution of these system call numbers to be considered an exception.

A "protected" exception port is one which cannot be fetched and for which exception processing cannot be aborted (**thread_abort**).

RETURN VALUE

KERN_EXCEPTION_PROTECTED
: One of the requested exception ports is protected and cannot be returned.

RELATED INFORMATION

Functions: **mach_thread_self**, **task_get_exception_ports**, **task_set_exception_ports**, **task_swap_exception_ports**, **thread_create**, **thread_get_exception_ports**, **thread_set_exception_ports**, **catch_exception_raise**, **thread_abort**.

thread_switch

Status:

Believed to be stable

SWITCH_OPTION_IDLE added **[OSF/1 R1.3]**.

System Trap — Cause context switch with options

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t thread_switch
        (thread_port_t          new_thread,
         switch_option_t        option,
         mach_msg_timeout_t     time);
```

DESCRIPTION

The **thread_switch** function provides low-level access to the scheduler's context switching code. *new_thread* is a hint that implements hand-off scheduling. The operating system will attempt to switch directly to the new thread (bypassing the normal logic that selects the next thread to run) if possible. Since this is a hint, it may be incorrect; it is ignored if it doesn't specify a thread on the same host as the current thread or if the scheduler cannot switch to that thread (i.e., not runable or already running on another processor). In this case, the normal logic to select the next thread to run is used; the current thread may continue running if there is no other appropriate thread to run.

The *option* argument specifies the interpretation and use of *time*. The possible values (from **<mach/thread_switch.h>**) are:

SWITCH_OPTION_NONE
: The *time* argument is ignored.

SWITCH_OPTION_WAIT
: The thread is blocked for the specified *time*. This wait cannot be canceled by **thread_resume**; only **thread_abort** can terminate this wait.

SWITCH_OPTION_DEPRESS
: The thread's scheduling attributes are temporarily set so as to provide it with the lowest possible service for duration *time*. The scheduling depression is aborted when *time* has passed, when the current thread is next run (either via hand-off scheduling or because the processor set has nothing better to do), or when **thread_abort** or **thread_depress_abort** is applied to the current thread. Changing the thread's scheduling attributes (via **thread_policy**) will not affect this depression.

SWITCH_OPTION_IDLE
: This option is similar to SWITCH_OPTION_DEPRESS however, the thread's scheduling attributes are temporarily set so as to place it at a

service level that is below all runnable threads for duration *time*. The scheduling depression is aborted when *time* has passed, when the current thread is next run (either via hand-off scheduling or because the processor set has nothing better to do), or when **thread_abort** or **thread_depress_abort** is applied to the current thread. Changing the thread's scheduling attributes (via **thread_policy**) will not affect this depression.

The minimum time and units of time can be obtained as the *min_timeout* value from the HOST_SCHED_INFO flavor of **host_info**.

PARAMETERS

new_thread
[in thread send right] Thread to which the processor should switch context.

option
[in scalar] Options applicable to the context switch.

time
[in scalar] Time duration during which the thread should be affected by *option*.

NOTES

thread_switch is often called when the current thread can proceed no further for some reason; the various options and arguments allow information about this reason to be transmitted to the kernel. The *new_thread* argument (hand-off 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 short, especially when the identity of the thread for which this thread must wait is not known.

CAUTIONS

Users should beware of calling **thread_switch** with an invalid hint (e.g., THREAD_NULL) and no option. Because the time-sharing scheduler varies 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.

thread_switch ignores policies. Users relying on the preemption semantics of a fixed time policy should be aware that **thread_switch** ignores these semantics; it will run the specified *new_thread* independent of its scheduling attributes and those of any threads that could run instead.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **thread_abort**, **thread_depress_abort**.

Status:

Believed to be stable

thread_terminate

Function — Destroys a thread

SYNOPSIS

kern_return_t **thread_terminate**
(thread_port_t *target_thread)*;

DESCRIPTION

The **thread_terminate** function kills *target_thread*.

PARAMETERS

target_thread
[in thread send right] The thread to be destroyed.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_terminate**, **task_threads**, **thread_create**, **thread_resume**, **thread_suspend**.

thread_wire

Status:

Believed to be stable

Function — Marks the thread as privileged with respect to kernel resources

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t thread_wire
                (thread_port_t                    thread,
                 host_control_port_t              host_priv,
                 boolean_t                        wired);
```

DESCRIPTION

The **thread_wire** function marks the thread as “wired”. A “wired” thread is always eligible to be scheduled and can consume physical memory even when free memory is scarce. This property should be assigned to threads in the default page-out path. Threads not in the default page-out path should not have this property to prevent the kernel’s free list of pages from being exhausted.

PARAMETERS

thread
: [in thread send right] The thread to be wired.

host_priv
: [in host-control send right] The privileged control port for the host on which the thread executes.

wired
: [in scalar] TRUE if the thread is to be wired.

RETURN VALUE

KERN_INVALID_HOST
: *host_priv* is not the control port for the host on which *thread* executes.

RELATED INFORMATION

Functions: **vm_wire**.

CHAPTER 7

Task Interface

This chapter discusses the specifics of the kernel's task interfaces. This includes functions that return status information for a task. Also included are functions that operate upon all or a set of threads within a task.

Status:

Believed to be stable

mach_ports_lookup

Function — Returns an array of well-known system ports.

SYNOPSIS

kern_return_t **mach_ports_lookup**
(task_port_t *target_task,*
mach_port_array_t* *init_port_set,*
mach_msg_type_number_t* *init_port_count)*;

DESCRIPTION

The **mach_ports_lookup** function returns an array of the well-known system ports that are currently registered for the specified task. Note that the task holds only send rights for the ports.

Registered ports are those ports that are used by the run-time system to initialize a task. To register system ports for a task, use the **mach_ports_register** function.

PARAMETERS

target_task
[in task send right] The task whose currently registered ports are to be returned.

init_port_set
[out pointer to dynamic array of registered send rights] The returned array of ports.

init_port_count
[out scalar] The number of returned port rights.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_ports_register**.

mach_ports_register

Status:

Believed to be stable

Function — Registers an array of well-known system ports

SYNOPSIS

kern_return_t **mach_ports_register**

(task_port_t	*target_task,*
mach_port_array_t	*init_port_set,*
mach_msg_type_number_t	*init_port_array_count)*;

DESCRIPTION

The **mach_ports_register** function registers an array of well-known system ports for the specified task. The task holds only send rights for the registered ports. The valid well-known system ports are:

- The port for the Name Server.
- The port for the Environment Manager.
- The port for the Service server.

Each port must be placed in a specific slot in the array. The slot numbers are defined (in **mach.h**) by the global constants NAME_SERVER_SLOT, ENVIRONMENT_SLOT, and SERVICE_SLOT.

A task can retrieve the currently registered ports by using the **mach_ports_lookup** function.

PARAMETERS

target_task
: [in task send right] The task for which the ports are to be registered.

init_port_set
: [in pointer to array of registered send rights] The array of ports to register.

init_port_array_count
: [in scalar] The number of ports in the array. Note that while this is a variable, the kernel accepts only a limited number of ports. The maximum number of ports is defined by the global constant MACH_PORTS_SLOTS_USED.

NOTES

When a new task is created (with **task_create**), the child task can inherit the parent's registered ports. Note that child tasks do not automatically acquire rights to these ports. They must use **mach_ports_lookup** to get them. It is intended that port registration be used only for task initialization, and then only by run-time support modules.

A parent task has three choices when passing registered ports to child tasks:

- The parent task can do nothing. In this case, all child tasks inherit access to the same ports that the parent has.
- The parent task can use **mach_ports_register** to modify its set of registered ports before creating child tasks. In this case, the child tasks get access to the modified set of ports. After creating its child tasks. the parent can use **mach_ports_register** again to reset its registered ports.
- The parent task can first create a specific child task and then use **mach_-ports_register** to modify the child's inherited set of ports, before starting the child's thread(s). The parent must specify the child's task port, rather than its own, on the call to **mach_ports_register**.

Tasks other than the Name Server and the Environment Manager should not need access to the Service port. The Name Server port is the same for all tasks on a given machine. The Environment port is the only port likely to have different values for different tasks.

Registered ports are restricted to those ports that are used by the run-time system to initialize a task. A parent task can pass other ports to its child tasks through:

- An initial message (see **mach_msg**).
- The Name Server, for public ports.
- The Environment Manager, for private ports.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_msg**, **mach_ports_lookup**.

mach_task_self

Status:

Believed to be stable

System Trap — Returns the task self port

LIBRARY

#include **<mach/mach_traps.h>**

SYNOPSIS

```
task_port_t mach_task_self
                ();
```

DESCRIPTION

The **mach_task_self** function returns send rights to the task's kernel port.

PARAMETERS

None

NOTES

The include file **<mach_init.h>** included by **<mach.h>** redefines this function call to simply return the value **mach_task_self_**, cached by the Mach run-time.

RETURN VALUE

[task-self send right] Send rights to the task's port.

RELATED INFORMATION

Functions: **task_set_special_port**.

task_create

Status:

Believed to be stable

Function — Creates a task

SYNOPSIS

kern_return_t **task_create**

(task_port_t	*parent_task,*
boolean_t	*inherit_memory,*
task_port_t*	*child_task*);

DESCRIPTION

The **task_create** function creates a new task from *parent_task* and returns the name of the new task in *child_task*. The child task acquires shared or copied parts of the parent's address space (see **vm_inherit**). The child task initially contains no threads. The child task inherits the parent's security ID.

The child task receives the following "special" ports, which are created or copied for it at task creation:

- [task-self send right] The port by which the kernel knows the new child task and allows it to be manipulated. The child task holds a send right for this port. The port name is also returned to the calling task.
- [bootstrap send right] The port to which the child task can send a message requesting return of any system service ports that it needs (for example, a port to the Network Name Server or the Environment Manager). The child task inherits a send right for this port from the parent task. The task can use **task_set_special_port** to change this port.
- [host-self send right] The port by which the child task requests information about its host. The child task inherits a send right for this port from the parent task.

The child task also inherits the following ports:

- [sample send right] The port to which PC sampling messages are to be sent.
- [exception send rights] Ports to which exception messages are sent.
- [registered send rights] Ports to system services.

PARAMETERS

parent_task
: [in task send right] The port for the task from which to draw the child task's port rights and address space.

inherit_memory
[in scalar] Address space inheritance indicator. If true, the child task inherits the (inheritable) address space of the parent task. If false, the kernel assigns the child task an empty address space.

child_task
[out task send right] The kernel-assigned port for the new task.

RETURN VALUE

Generic errors apply.

RELATED INFORMATION

Functions: **task_create_security_id**, **task_resume**, **task_set_special_port**, **task_suspend**, **task_terminate**, **task_threads**, **thread_create**, **thread_resume**, **vm_inherit**, **task_sample**, **task_set_exception_ports**, **mach_ports_register**, **norma_task_create**, **task_set_security_id**.

Status:

Believed to be stable

task_create_security_id

Function — Creates a new task with an explicit security ID

SYNOPSIS

kern_return_t **task_create_security_id**

(task_port_t	*task,*
security_port_t	*host_security,*
security_id_t	*security_id*,
boolean_t	*inherit_memory,*
task_port_t*	*child_task*);

DESCRIPTION

The **task_create_security_id** function creates a new task from *parent_task* with an explicit security ID, returning the name of the new task in *child_task*. Other than the security ID, the child task is as if created by **task_create**.

PARAMETERS

parent_task
[in task send right] The port for the task from which to draw the child task's port rights and address space.

host_security
[in security send right] The host's security port.

security_id
[in scalar] The task's security ID.

inherit_memory
[in scalar] Address space inheritance indicator. If true, the child task inherits the address space of the parent task. If false, the kernel assigns the child task an empty address space.

child_task
[out task send right] The kernel-assigned port name for the new task.

NOTES

The host security port is a privileged port given to the system bootstrap task for the use of this call.

RETURN VALUE

KERN_INVALID_SECURITY
host_security is not the security port for the host on which *task* lies.

RELATED INFORMATION

Functions: **task_create**, **task_set_security_id**, **mach_msg**.

Status:

Believed to be stable

task_get_emulation_vector

Function — Return user-level handlers for system calls.

SYNOPSIS

```
kern_return_t task_get_emulation_vector
        (task_port_t                          task,
         syscall_number_t*                    vector_start,
         emulation_vector_t*                  emulation_vector,
         mach_msg_type_number_t*              emulation_vector_count);
```

DESCRIPTION

The **task_get_emulation_vector** function returns the user-level syscall handler entrypoint addresses.

PARAMETERS

task
: [in task send right] The port for the task for which the system call handler addresses are desired.

vector_start
: [out scalar] The syscall number corresponding to the first element of *emulation_vector*.

emulation_vector
: [out pointer to dynamic array of *vm_address_t*] Pointer to the returned array of routine entrypoints for the system calls starting with syscall number *vector_start*.

emulation_vector_count
: [out scalar] The number of entries filled by the kernel.

NOTES

This interface is machine word length specific because of the virtual addresses in the *emulation_vector* parameter.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_set_emulation_vector**.

task_get_exception_ports

Status:

Believed to be stable

Function — Returns send rights to exception ports

SYNOPSIS

kern_return_t **task_get_exception_ports**

(task_port_t	*task,*
exception_mask_t	*exception_types*,
exception_mask_array_t	*old_exception_masks*,
mach_msg_type_number_t*	*old_exception_count*,
exception_port_array_t	*old_exception_ports*,
exception_behavior_array_t	*old_behaviors*,
thread_state_flavor_array_t	*old_flavors*);

DESCRIPTION

The **task_get_exception_ports** function returns send rights for a specified set of exception ports belonging to *task*. A task exception port is used when a thread specific exception port returns a non-success reply. The call returns a set of quadruples <exception type mask, exception port, behavior, flavor> for each unique set of <exception port, behavior, flavor> in effect for the task where the exception type mask indicates for which exception types the other values apply.

PARAMETERS

task
: [in task send right] The task for which to return the exception ports.

exception_types
: [in scalar] A flag word indicating the types of exceptions for which the exception ports are desired.

EXC_MASK_BAD_ACCESS
: Could not access memory.

EXC_MASK_BAD_INSTRUCTION
: Instruction failed. Illegal or undefined instruction or operand.

EXC_MASK_ARITHMETIC
: Arithmetic exception

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
: Trace, breakpoint, etc.

EXC_MASK_SYSCALL
: System call requested.

EXC_MASK_MACH_SYSCALL
: System call with a number in the Mach call range requested.

old_exception_masks
: [out array of *exception_mask_t*] An array, each element being a mask specifying for which exception types the corresponding element of the other arrays apply.

old_exception_count
: [pointer to in/out scalar] On input, the maximum size of the array buffers; on output, the number of returned <exception type mask, exception port, behavior, flavor> sets returned.

old_exception_ports
: [out array of exception send rights] The returned exception ports.

old_behaviors
: [out array of *exception_behavior_t*] The type of exception message to be sent. Defined types are:

EXCEPTION_DEFAULT
: Send a **catch_exception_raise** message including the thread identity.

EXCEPTION_STATE
: Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_IDENTITY
: Send a **catch_exception_raise_state_identity** message including the thread identity and state.

old_flavors
: [out array of *thread_state_flavor_t*] The type of state to be sent with the exception message. These types are defined in **<mach/thread_states.h>**.

RETURN VALUE

KERN_EXCEPTION_PROTECTED
: One of the requested exception ports is protected and cannot be returned.

RELATED INFORMATION

Functions: **mach_task_self**, **thread_get_exception_ports**, **task_set_exception_ports**, **task_swap_exception_ports**, **thread_create**, **thread_set_exception_ports**, **thread_swap_exception_ports**, **catch_exception_raise**.

Status:

Believed to be stable

task_get_special_port

Function — Returns a send right to a special port

SYNOPSIS

kern_return_t **task_get_special_port**
(task_port_t *task,*
task_special_port_t *which_port,*
mach_port_t* *special_port*);

task_get_bootstrap_port

Macro form

kern_return_t **task_get_bootstrap_port**
(task_port_t *task,*
mach_port_t* *special_port)*

⇒ **task_get_special_port** (*task*, TASK_BOOTSTRAP_PORT, *special_port*)

task_get_kernel_port

Macro form

kern_return_t **task_get_kernel_port**
(task_port_t *task,*
mach_port_t* *special_port)*

⇒ **task_get_special_port** (*task*, TASK_KERNEL_PORT, *special_port*)

task_get_host_name_port

Macro form

kern_return_t **task_get_host_name_port**
(task_port_t *task,*
mach_port_t* *special_port)*

⇒ **task_get_special_port** (*task*, TASK_HOST_NAME_PORT, *special_port*)

DESCRIPTION

The **task_get_special_port** function returns a send right for a special port belonging to *task*.

If one task has a send right for the kernel port of another task, it can use the port to perform kernel operations for the other task. Send rights for a kernel port normally are held only by the task to which the port belongs, or by the task's parent task. Using the **mach_msg** function, however, any task can pass a send right for its kernel port to another task.

PARAMETERS

task
: [in task send right] The port for the task for which to return the port's send right.

which_port
: [in scalar] The special port for which the send right is requested. Valid values are:

TASK_KERNEL_PORT
: [task-self send right] The port used to control this task. Used to send messages that affect the task. This is the port returned by **mach_task_self**.

TASK_BOOTSTRAP_PORT
: [bootstrap send right] The task's bootstrap port. Used to send messages requesting return of other system service ports.

TASK_HOST_NAME_PORT
: [host-self send right] The port used to request information of the containing host. This is the port returned by **mach_host_self**.

special_port
: [out task-special send right] The returned value for the port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_task_self**, **task_create**, **task_set_special_port**, **thread_get_special_port**, **thread_set_special_port**, **mach_host_self**.

Status:

Believed to be stable

task_info

Function — Returns information about a task

SYNOPSIS

```
kern_return_t task_info
        (task_port_t                         task,
         task_flavor_t                       flavor,
         task_info_t                         task_info,
         mach_msg_type_number_t*             task_info_count);
```

DESCRIPTION

The **task_info** function returns an information structure of type *flavor*.

PARAMETERS

task
: [in task send right] The port for the task for which the information is to be returned.

flavor
: [in scalar] The type of information to be returned. Valid values are:

TASK_BASIC_INFO
: Returns basic information about the task, such as the task's suspend count and number of resident pages. The structure returned is **task_basic_info**.

TASK_THREAD_TIMES_INFO
: Returns system and user space run-times for live threads. The structure returned is **task_thread_times_info**.

TASK_SCHED_FIFO_INFO
: Returns default FIFO scheduling policy attributes to be assigned to new threads. The structure returned is **policy_fifo_base**.

TASK_SCHED_RR_INFO
: Returns default round-robin scheduling policy attributes to be assigned to new threads. The structure returned is **policy_rr_base**.

TASK_SCHED_TIMESHARE_INFO
: Returns default timeshare scheduling policy attributes to be assigned to new threads. The structure returned is **policy_timeshare_base**.

TASK_SECURITY_ID
Returns the security ID for the task. The value returned is of type *security_id_t*

task_info
[out structure] Information about the specified task.

task_info_count
[in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

NOTES

At any given time, a task has one default scheduling policy assigned to it (as returned by TASK_BASIC_INFO). As such, only one of the scheduling flavors will return valid information.

RETURN VALUE

KERN_INVALID_POLICY
A request was made for the default scheduling policy attributes for the task but the requested policy is not the task's default policy.

RELATED INFORMATION

Functions: **task_get_special_port**, **task_set_special_port**, **task_threads**, **thread_info**, **thread_get_state**, **thread_set_state**, **task_set_attributes**.

Data Structures: **task_basic_info**, **policy_timeshare_info**, **policy_fifo_info**, **policy_rr_info**, **task_thread_times_info**.

Status:

Believed to be stable

task_resume

Function — Resume a task

SYNOPSIS

kern_return_t **task_resume**
(task_port_t *task*);

DESCRIPTION

The **task_resume** function decrements the suspend count for *task*. If the task's suspend count goes to zero, the function resumes any suspended threads within the task. To resume a given thread, the thread's own suspend count must also be zero.

PARAMETERS

task
[in task send right] The port to the task to be resumed.

NOTES

An attempt to lower the suspend count below zero is ignored.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_info**, **task_suspend**, **task_terminate**, **thread_info**, **thread_resume**, **thread_suspend**.

task_sample

Status:

Experimental support for the profil system call

Function — Perform periodic PC sampling for a task

SYNOPSIS

```
kern_return_t task_sample
        (task_port_t            sample_task,
         sample_port_t          reply_port);
```

DESCRIPTION

The **task_sample** function causes the program counter (PC) of the specified *sample_task* (actually, all of the threads within *sample_task*) to be sampled periodically (whenever one of the threads happens to be running at the time of the kernel's "hardclock" interrupt). The set of PC sample values obtained are saved in buffers which are sent to the specified *reply_port* in **receive_samples** messages.

PARAMETERS

sample_task
: [in task send right] Port for the task whose threads' PC are to be sampled

reply_port
: [in sample receive (to be converted to send) right] Port to which PC sample buffers are sent. A value of MACH_PORT_NULL stops PC sampling for the task.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **thread_sample**, **receive_samples**.

Status:

Believed to be stable

task_set_emulation_vector

Function — Establishes user-level handlers for system calls.

SYNOPSIS

kern_return_t **task_set_emulation_vector**
(task_port_t *task,*
syscall_number_t *vector_start*,
emulation_vector_t *emulation_vector,*
mach_msg_type_number_t *emulation_vector_count*);

DESCRIPTION

The **task_set_emulation_vector** function establishes a handler within the task for a set of system calls. When a thread executes a system call with one of these numbers, the system call will be redirected to the corresponding routine within the task's address space.

These emulation handler addresses are inherited by child processes.

PARAMETERS

task
: [in task send right] The port for the task for which to establish the system call handler.

vector_start
: [in scalar] The syscall number corresponding to the first element of *emulation_vector.*

emulation_vector
: [pointer to in array of *vm_address_t*] An array of routine entrypoints for the system calls starting with syscall number *vector_start*.

emulation_vector_count
: [in scalar] The number of elements in *emulation_vector.*

NOTES

This interface is machine word length specific because of the virtual addresses in the *emulation_vector* parameter.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_get_emulation_vector**.

Status:

Believed to be stable

task_set_exception_ports

Function — Sets exception ports for a task

SYNOPSIS

kern_return_t **task_set_exception_ports**

(task_port_t	*task,*
exception_mask_t	*exception_types,*
exception_port_t	*exception_port*,
exception_behavior_t	*behavior*,
thread_state_flavor_t	*flavor*);

DESCRIPTION

The **task_set_exception_ports** function sets a specified set of exception ports belonging to *task*. A task exception port is used when a thread specific exception port returns a non-success reply.

PARAMETERS

task
[in task send right] The task for which to set the ports.

exception_types
[in scalar] A flag word indicating the types of exceptions for which the exception port applies.

EXC_MASK_BAD_ACCESS
Could not access memory.

EXC_MASK_BAD_INSTRUCTION
Instruction failed. Illegal or undefined instruction or operand.

EXC_MASK_ARITHMETIC
Arithmetic exception

EXC_MASK_EMULATION
Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
Software generated exception.

EXC_MASK_BREAKPOINT
Trace, breakpoint, etc.

EXC_MASK_SYSCALL
System call requested.

EXC_MASK_MACH_SYSCALL
System call with a number in the Mach call range requested.

exception_port
[in exception send right] The exception port for all selected exception types.

behavior
[in scalar] The type of exception message to be sent. Defined types are:

EXCEPTION_DEFAULT
Send a **catch_exception_raise** message including the thread identity.

EXCEPTION_DEFAULT_PROTECTED
Send a **catch_exception_raise** message including the thread identity. Mark the exception port (and associated exceptions) as protected.

EXCEPTION_STATE
Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_PROTECTED
Send a **catch_exception_raise_state** message including the thread state. Mark the exception port (and associated exceptions) as protected.

EXCEPTION_STATE_IDENTITY
Send a **catch_exception_raise_state_identity** message including the thread identity and state.

EXCEPTION_STATE_IDENTITY_PROTECTED
Send a **catch_exception_raise_state_identity** message including the thread identity and state. Mark the exception port (and associated exceptions) as protected.

flavor
[in scalar] The type of state to be sent with the exception message. These types are defined in **<mach/thread_states.h>**.

NOTES

If the value of the EXC_MACH_SYSCALL exception class exception port is the host name port, Mach kernel traps are executed by the kernel as expected;

any other value causes the attempted execution of these system call numbers to be considered an exception.

A “protected” exception port is one which cannot be fetched and for which exception processing cannot be aborted (**thread_abort**).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_task_self**, **task_get_exception_ports**, **task_swap_exception_ports**, **thread_set_exception_ports**, **thread_create**, **thread_get_exception_ports**, **thread_swap_exception_ports**, **catch_exception_raise**, **thread_abort**.

task_set_security_id

Status:

Believed to be stable

Function — Changes a task's security ID

SYNOPSIS

```
kern_return_t task_set_security_id
        (task_port_t                 task,
         security_port_t    host_security,
         security_id_t        security_id);
```

DESCRIPTION

The **task_set_security_id** function changes the security ID for the task included in subsequent messages sent by the task. The initial value of a task's security ID is that of its parent.

PARAMETERS

task
: [in task send right] The port for the task for which the ID is to be set.

host_security
: [in security send right] The host's security port.

security_id
: [in scalar] The new security ID.

NOTES

The host security port is a privileged port given to the system bootstrap task for the use of this call.

RETURN VALUE

KERN_INVALID_SECURITY
: *host_security* is not the security port for the host on which *task* lies.

RELATED INFORMATION

Functions: **task_create**, **task_info**, **mach_msg**.

Status:

Believed to be stable

task_set_special_port

Function — Sets a special port for a task

SYNOPSIS

kern_return_t **task_set_special_port**
(task_port_t *task,*
task_special_port_t *which_port,*
mach_port_t *special_port*);

task_set_bootstrap_port

Macro form

kern_return_t **task_set_bootstrap_port**
(task_port_t *task,*
mach_port_t *special_port)*

⇒ **task_set_special_port** (*task*, TASK_BOOTSTRAP_PORT, *special_port*)

task_set_kernel_port

Macro form

kern_return_t **task_set_kernel_port**
(task_port_t *task,*
mach_port_t *special_port)*

⇒ **task_set_special_port** (*task*, TASK_KERNEL_PORT, *special_port*)

task_set_host_name_port

Macro form

kern_return_t **task_set_host_name_port**
(task_port_t *task,*
mach_port_t *special_port)*

⇒ **task_set_special_port** (*task*, TASK_HOST_NAME_PORT, *special_port*)

DESCRIPTION

The **task_set_special_port** function sets a special port belonging to *task*.

PARAMETERS

task
[in task send right] The port for the task for which to set the port.

which_port
[in scalar] The special port to be set. Valid values are:

TASK_BOOTSTRAP_PORT
[bootstrap send right] The task's bootstrap port. Used to send messages requesting return of other system service ports.

TASK_KERNEL_PORT
[task-self send right] The task's kernel port. Used by the kernel to receive messages to manipulate the task. This is the port returned by **mach_task_self**. Setting this special port does not change the identity of the kernel port that names the task; this simply changes the value returned as the kernel special port.

TASK_HOST_NAME_PORT
[host-self send right] The task's host self port. Used by the task to request information about its containing host. This is the port returned by **mach_host_self**. Setting this special port does not change the identity of the kernel port that names the host; this simply changes the value returned as the host special port.

special_port
[in task-special send right] The value for the port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_get_special_port**, **mach_task_self**, **thread_get_special_port**, **thread_set_special_port**, **mach_host_self**.

Status:

Believed to be stable

task_suspend

Function — Suspends a task

SYNOPSIS

kern_return_t **task_suspend**
    (task_port_t *task*);

DESCRIPTION

The **task_suspend** function increments the suspend count for *task* and stops all threads within the task. As long as the suspend count is positive, no newly-created threads can execute. The function does not return until all of the task's threads have been suspended.

PARAMETERS

task
    [in task send right] The port for the task to be suspended.

NOTES

To resume a suspended task and its threads, use **task_resume**. If the suspend count is greater than one, **task_resume** must be repeated that number of times.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_info**, **task_resume**, **task_terminate**, **thread_suspend**.

task_swap_exception_ports

Status:

Believed to be stable

Function — Swaps exception ports for a task

SYNOPSIS

kern_return_t **task_swap_exception_ports**

(task_port_t	*task,*
exception_mask_t	*exception_types,*
exception_port_t	*exception_port*,
exception_behavior_t	*behavior*,
thread_state_flavor_t	*flavor*,
exception_mask_array_t	*old_exception_masks*,
mach_msg_type_number_t*	*old_exception_count*,
exception_port_array_t	*old_exception_ports*,
exception_behavior_array_t	*old_behaviors*,
thread_state_flavor_array_t	*old_flavors*);

DESCRIPTION

The **task_swap_exception_ports** function sets a specified set of exception ports belonging to *task*, returning the old set. A task exception port is used when a thread specific exception port returns a non-success reply.

PARAMETERS

task
: [in task send right] The task for which to set the ports.

exception_types
: [in scalar] A flag word indicating the types of exceptions for which the exception port applies.

EXC_MASK_BAD_ACCESS
: Could not access memory.

EXC_MASK_BAD_INSTRUCTION
: Instruction failed. Illegal or undefined instruction or operand.

EXC_MASK_ARITHMETIC
: Arithmetic exception

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
: Trace, breakpoint, etc.

EXC_MASK_SYSCALL
: System call requested.

EXC_MASK_MACH_SYSCALL
: System call with a number in the Mach call range requested.

exception_port
: [in exception send right] The exception port for all selected exception types.

behavior
: [in scalar] The type of exception message to be sent. Defined types are:

EXCEPTION_DEFAULT
: Send a **catch_exception_raise** message including the thread identity.

EXCEPTION_DEFAULT_PROTECTED
: Send a **catch_exception_raise** message including the thread identity. Mark the exception port (and associated exceptions) as protected.

EXCEPTION_STATE
: Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_PROTECTED
: Send a **catch_exception_raise_state** message including the thread state. Mark the exception port (and associated exceptions) as protected.

EXCEPTION_STATE_IDENTITY
: Send a **catch_exception_raise_state_identity** message including the thread identity and state.

EXCEPTION_STATE_IDENTITY_PROTECTED
: Send a **catch_exception_raise_state_identity** message including the thread identity and state. Mark the exception port (and associated exceptions) as protected.

flavor
: [in scalar] The type of state to be sent with the exception message. These types are defined in **<mach/thread_states.h>**.

old_exception_masks
: [out array of *exception_mask_t*] An array, each element being a mask specifying for which exception types the corresponding element of the other arrays apply.

old_exception_count
: [pointer to in/out scalar] On input, the maximum size of the array buffers; on output, the number of returned <exception type mask, exception port, behavior, flavor> sets returned.

old_exception_ports
: [out array of exception send rights] The returned exception ports.

old_behaviors
: [out array of *exception_behavior_t*] The type of exception message to be sent as with *behavior*.

old_flavors
: [out array of *thread_state_flavor_t*] The type of state to be sent with the exception message. These types are defined in **<mach/thread_states.h>**.

NOTES

If the value of the EXC_MACH_SYSCALL exception class exception port is the host name port, Mach kernel traps are executed by the kernel as expected; any other value causes the attempted execution of these system call numbers to be considered an exception.

A “protected” exception port is one which cannot be fetched and for which exception processing cannot be aborted (**thread_abort**).

RETURN VALUE

KERN_EXCEPTION_PROTECTED
: One of the requested exception ports is protected and cannot be returned.

RELATED INFORMATION

Functions: **mach_task_self**, **task_get_exception_ports**, **task_set_exception_ports**, **thread_create**, **thread_get_exception_ports**, **thread_set_exception_ports**, **thread_swap_exception_ports**, **catch_exception_raise**, **thread_abort**.

Status:

Believed to be stable

task_terminate

Function — Destroys a task

SYNOPSIS

kern_return_t **task_terminate**
(task_port_t *task*);

DESCRIPTION

The **task_terminate** function kills *task* and all its threads, if any. The kernel frees all resources that are in use by the task. The kernel destroys any port for which the task holds the receive right.

PARAMETERS

task
[in task send right] The port for the task to be destroyed.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_suspend**, **task_resume**, **thread_terminate**, **thread_suspend**.

task_threads

Status:

Believed to be stable

Function — Returns a list of the threads within a task

SYNOPSIS

kern_return_t **task_threads**

(task_port_t	*task,*
thread_port_array_t*	*thread_list,*
mach_msg_type_number_t*	*thread_count*);

DESCRIPTION

The **task_threads** function returns a list of the threads within *task*. The calling task or thread also receives a send right to the kernel port for each listed thread.

PARAMETERS

task
: [in task send right] The port for the task for which the thread list is to be returned.

thread_list
: [out pointer to dynamic array of thread send rights] The returned list of threads within *task*, in no particular order.

thread_count
: [out scalar] The returned count of threads in *thread_list*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **thread_create**, **thread_terminate**, **thread_suspend**.

CHAPTER 8

Host, Ledger and Clock Interface

This chapter discusses the specifics of the kernel's host interfaces. Included are functions that return status information for a host, such as kernel statistics.

Note that hosts are named both by a name port, which allows the holder to request information about the host, and a control port, which provides full control access.

Associated with each host is a (small) set of clocks used to keep track of time and to generate alarms at requested times. The kernel clock objects are also named by both a name and control port.

Status:

Believed to be stable

clock_alarm

Function — Request a clock alarm

SYNOPSIS

```
kern_return_t clock_alarm
        (clock_name_port_t        clock_name,
         alarm_type_t             alarm_type,
         tvalspec_t               alarm_time,
         mach_port_t              alarm_reply_port);
```

DESCRIPTION

The **clock_alarm** function requests that a clock send an alarm message to a specified port at a given future time. The alarm message is specified by the **clock_alarm_reply** server interface.

PARAMETERS

clock_name
: [in clock-name send right] The name (or control) port for the clock.

alarm_type
: [in scalar] How to interpret the *alarm_time* value:

  TIME_RELATIVE
  : Interpret the alarm time as relative to the current time.

  TIME_ABSOLUTE
  : Interpret the alarm time as an absolute time.

alarm_time
: [in structure] The time when the alarm is to be sent.

alarm_reply_port
: [in alarm receive (to be converted to send-once) right] A port into which the alarm message is to be sent.

NOTES

If the specified alarm time is in the past, the alarm message is sent immediately and time-stamped with the current time. Otherwise, the alarm is queued and delivered at the specified alarm time and time-stamped at that time.

The alarm will be serviced at the service time nearest the specified alarm time as governed by the current clock alarm resolution.

Not all clocks implement this service, but the REALTIME clock must. If the clock does not provide this service, this call is ignored.

RETURN VALUE

Since this is an asynchronous function that does not wait for a reply, only message transmission errors apply.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **clock_sleep**, **clock_alarm_reply**.

Data Structures: **tvalspec**.

Status:

Believed to be stable

clock_alarm_reply

Server Interface — Indicates that an alarm went off

SYNOPSIS

```
kern_return_t clock_alarm_reply
        (mach_port_t         alarm_reply_port,
         kern_return_t       reply_code,
         alarm_type_t        alarm_type,
         tvalspec_t          wake_time);
```

DESCRIPTION

A **clock_alarm_reply** function is called as the result of a message from the kernel indicating that a previously requested alarm time (**clock_alarm**) has arrived.

PARAMETERS

alarm_reply_port
[in alarm (receive) right] The reply port named in the corresponding **clock_alarm** call.

reply_code
[in scalar] The reply status code from the alarm. The possible values are:

KERN_SUCCESS
The alarm was delivered without problem.

KERN_INVALID_VALUE
The *alarm_type* and/or *alarm_time* values supplied to **clock_alarm** were invalid.

KERN_INVALID_LEDGER
The *ledger* supplied to **clock_alarm** was not a ledger.

KERN_ABORTED
The alarm was terminated via use of **clock_set_time**.

alarm_type
[in scalar] The alarm type value supplied to the **clock_alarm** call. Only the low order bits of this value are used by the kernel so the high order bits are available for application use as an alarm identifier.

wake_time
[in structure] The time when the alarm message was sent.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **clock_sleep**, **clock_alarm**, **clock_reply_server**, **clock_set_time**.

Data Structures: **tvalspec**.

Status:

Believed to be stable

clock_get_attributes

Function — Returns attributes of a clock

SYNOPSIS

kern_return_t **clock_get_attributes**

(clock_name_port_t	*clock_name,*
clock_flavor_t	*flavor,*
clock_attr_t	*attribute*,
mach_msg_type_number_t*	*attribute_count*);

DESCRIPTION

The **clock_get_attributes** function returns attributes of a clock's implementation or operation.

PARAMETERS

clock_name
[in clock-name send right] The name (or control) port for the clock.

flavor
[in scalar] Type of information desired. Defined values are:

CLOCK_GET_TIME_RES
The resolution, in nanoseconds, with which the value returned by **clock_get_time** is updated.

CLOCK_MAP_TIME_RES
The resolution, in nanoseconds, with which the value visible via **clock_map_time** is updated.

CLOCK_ALARM_CURRES
The resolution, in nanoseconds, at which clock alarm and sleep timers are currently serviced.

CLOCK_ALARM_MINRES
The minimum resolution, in nanoseconds, at which clock alarm and sleep timers can be serviced.

CLOCK_ALARM_MAXRES
The maximum resolution, in nanoseconds, at which clock alarm and sleep timers can be serviced.

attribute
[out scalar] The returned attribute.

attribute_count
[in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_time**, **clock_map_time**, **clock_-sleep**, **clock_alarm**, **clock_set_attributes**.

Status:

Believed to be stable

clock_get_time

Function — Returns the current time

SYNOPSIS

```
kern_return_t clock_get_time
        (clock_name_port_t          clock_name,
         tvalspec_t*                cur_time);
```

DESCRIPTION

The **clock_get_time** function returns the current time kept by a clock. The value returned is a monotonically increasing value (unless tampered with via the **clock_set_time** function).

PARAMETERS

clock_name
: [in clock-name send right] The name (or control) port for the clock.

cur_time
: [out structure] Current time

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_map_time**, **clock_sleep**, **clock_alarm**, **clock_set_time**.

Data Structures: **tvalspec**.

clock_map_time

Status:

Believed to be stable

Function — Returns a memory object that maps a clock

SYNOPSIS

```
kern_return_t clock_map_time
        (clock_name_port_t       clock_name,
         mem_obj_rep_port_t*     clock_memory);
```

DESCRIPTION

The **clock_map_time** function returns a memory object representative port representing read access to a memory object that contains (at offset zero) a mapped version of the clock time (structure **mapped_tvalspec**). The returned right is suitable as an argument for **vm_map**.

PARAMETERS

clock_name
: [in clock-name send right] The name (or control) port for the clock.

clock_memory
: [out memory-object-representative send right] Mapped clock time memory object representative.

NOTES

Not all clocks provide this service, but the REALTIME clock must.

RETURN VALUE

KERN_FAILURE
: The specified clock does not provide this service.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **clock_sleep**, **clock_alarm**.

Data Structures: **mapped_tvalspec**.

Status:

Believed to be stable

clock_set_attributes

Function — Sets attributes of a clock

SYNOPSIS

```
kern_return_t clock_set_attributes
        (clock_control_port_t           clock_control,
         clock_flavor_t                 flavor,
         clock_attr_t                   attribute,
         mach_msg_type_number_t         attribute_count);
```

DESCRIPTION

The **clock_set_attributes** function sets attributes of a clock's operation.

PARAMETERS

clock_control
[in clock-control send right] The control port for the clock.

flavor
[in scalar] Type of information to be set. Defined values are:

CLOCK_ALARM_CURRES
The resolution, in nanoseconds, at which clock alarm and sleep timers are currently serviced. Increasing the current resolution will have no impact on any pending clock alarms (i.e. they will go off as originally scheduled). Decreasing the current resolution will truncate any pending alarms to the granularity of the new current resolution. This value must be a multiple of the minimum resolution and not greater than the maximum resolution of the clock.

attribute
[pointer to in scalar] New attribute.

attribute_count
[in scalar] The size of the buffer (in natural-sized units).

NOTES

The main reason a clock's current resolution would not always equal its minimum resolution is because the overhead of sustaining the minimum resolution, when it is not needed by any existing alarm service client, may be prohibitive for a given hardware platform and underlying clock device.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_control**, **clock_set_time**, **clock_get_attributes**.

Status:

Believed to be stable

clock_set_time

Function — Sets the current time

SYNOPSIS

```
kern_return_t clock_set_time
        (clock_control_port_t      clock_control,
         tvalspec_t                new_time);
```

DESCRIPTION

The **clock_set_time** function sets the time kept by a clock. Setting the clock time will cause all pending clock alarms and sleeps to be terminated with timestamps set to the current clock time just prior to the new time being set with a return code of KERN_ABORTED.

PARAMETERS

clock_control
: [in clock-control send right] The control port for the clock.

new_time
: [in structure] New time

CAUTIONS

The use of this function is highly discouraged since it could affect the monotonically increasing nature of the clock.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_control**, **clock_set_attributes**, **clock_get_time**, **clock_alarm**, **clock_sleep**.

Data Structures: **tvalspec**.

clock_sleep

Status:

Believed to be stable

System Trap — Sleep until a given time

SYNOPSIS

```
kern_return_t clock_sleep
        (clock_name_port_t        clock_name,
         sleep_type_t             sleep_type,
         tvalspec_t               sleep_time,
         tvalspec_t*              wake_time);
```

DESCRIPTION

The **clock_sleep** system trap delays the invoking thread until a specified time. This sleep may be aborted by **thread_abort**.

PARAMETERS

clock_name
[in clock-name send right] The name (or control) port for the clock.

sleep_type
[in scalar] How to interpret the *sleep_time* value:

TIME_RELATIVE
Interpret the sleep time as relative to the current time.

TIME_ABSOLUTE
Interpret the sleep time as an absolute time.

sleep_time
[in structure] The time when the sleep is to terminate.

wake_time
[out structure] The actual (absolute) time at which the sleep terminated.

NOTES

If the specified time is in the past, the call returns immediately with the wake time being the current time.

If the clock's time is changed (**clock_set_time**), the sleep will be interrupted.

The thread will waken at the service time nearest the specified sleep time as governed by the current clock alarm resolution.

Not all clocks provide this service but the REALTIME clock must.

RETURN VALUE

KERN_FAILURE
The clock does not support a sleep service.

KERN_ABORTED
The sleep was interrupted by **thread_abor**t or terminated via use of **clock_set_time**.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **clock_alarm**, **clock_set_time**, **thread_abort**.

Data Structures: **tvalspec**.

host_get_boot_info

Status:

Believed to be stable

Function — Return operator boot information

SYNOPSIS

kern_return_t **host_get_boot_info**
(host_control_port_t *priv_host,*
kernel_boot_info_t *boot_info*);

DESCRIPTION

The **host_get_boot_info** function returns the boot-time information string supplied by the operator when *priv_host* was initialized. The constant KERNEL_-BOOT_INFO_MAX (in **mach/host_info.h**) should be used to dimension storage for the returned string.

PARAMETERS

priv_host
[in host-control send right] The control port for the host for which information is to be obtained.

boot_info
[out array of *char*] Character string providing the operator boot info

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info**.

Status:

Believed to be stable

host_get_clock_control

Function — Returns a send right to a kernel clock's control port

SYNOPSIS

```
kern_return_t host_get_clock_control
        (host_control_port_t          host_priv,
         clock_id_t                   id,
         clock_control_port_t*        clock_control);
```

DESCRIPTION

The **host_get_clock_control** function returns a send right to the control port for a kernel clock object. This right is used to set the clock's resolution and time.

PARAMETERS

host_priv
: [in host-control send right] The control port for the host owning the clock.

id
: [in scalar] The identification of the desired kernel clock. These values are defined in **<mach/clock_types.h>**. Although an implementation may define additional values, the following values are always defined (although only the REALTIME clock is required to be implemented):

REALTIME_CLOCK
: A moderate resolution clock service that (typically) tracks time since the system last boot.

BATTERY_CLOCK
: A (typically) low resolution clock (to the second) that survives power failures or service outages.

HIGHRES_CLOCK
: A high resolution clock.

clock_control
: [out clock-control send right] Control port for the clock.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **clock_set_time**, **clock_set_attributes**, **host_get_clock_service**.

Status:

Believed to be stable

host_get_clock_service

Function — Returns a send right to a kernel clock's service port

SYNOPSIS

kern_return_t **host_get_clock_service**

(host_name_port_t	*host,*
clock_id_t	*id,*
clock_name_port_t*	*clock_name*);

DESCRIPTION

The **host_get_clock_service** function returns a send right to the name port for a kernel clock object. This right is used to get the time and resolutions of the clock and to set clock alarms.

PARAMETERS

host
: [in host-name send right] The name (or control) port for the host owning the clock.

id
: [in scalar] The identification of the desired kernel clock. These values are defined in **<mach/clock_types.h>**. Although an implementation may define additional values, the following values are always defined (although only the REALTIME clock is required to be implemented):

REALTIME_CLOCK
: A moderate resolution clock service that (typically) tracks time since the system last boot.

BATTERY_CLOCK
: A (typically) low resolution clock (to the second) that survives power failures or service outages.

HIGHRES_CLOCK
: A high resolution clock.

clock_name
: [out clock-name send right] Name port for the clock.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **clock_get_time**, **clock_get_attributes**, **clock_map_time**, **clock_-sleep**, **clock_alarm**, **host_get_clock_control**.

Status:

Believed to be stable

host_info

Function — Returns information about a host

SYNOPSIS

```
kern_return_t host_info
        (host_name_port_t                 host,
         host_flavor_t                    flavor,
         host_info_t                      host_info,
         mach_msg_type_number_t*          host_info_count);
```

DESCRIPTION

The **host_info** function returns selected information about a host, as specified by *flavor*.

PARAMETERS

host
: [in host-name send right] The name (or control) port for the host for which information is to be obtained.

flavor
: [in scalar] The type of statistics desired.

  HOST_BASIC_INFO
  : Basic information (number of processors, amount of memory). The returned structure is **host_basic_info**.

  HOST_SCHED_INFO
  : Basic restrictions of the kernel's scheduling, minimum quantum and time-out value. The returned structure is **host_sched_info**

  HOST_RESOURCE_SIZES
  : Size of significant kernel structures, as a ledger would consider them when limiting kernel resource consumption. The returned structure is **kernel_resource_sizes**.

host_info
: [out structure] Statistics about the specified host.

host_info_count
: [in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

NOTES

This interface is machine word length specific because of the memory size returned by HOST_BASIC_INFO.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_boot_info**, **host_kernel_version**, **host_statistics**.

Data Structures: **host_basic_info, host_sched_info**, **kernel_resource_sizes**.

host_kernel_version

Status:

Believed to be stable

Function — Returns kernel version information for a host

SYNOPSIS

kern_return_t **host_kernel_version**
(host_name_port_t *host,*
kernel_version_t *version*);

DESCRIPTION

The **host_kernel_version** function returns the version string compiled into the kernel executing on *host* at the time it was built. This describes the version of the kernel. The constant KERNEL_VERSION_MAX (in **mach/host_info.h**) should be used to dimension storage for the returned string if the *kernel_version_t* declaration is not used.

PARAMETERS

host
: [in host-name send right] The name (or control) port for the host for which information is to be obtained.

version
: [out array of *char*] Character string describing the kernel version executing on *host*

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info**.

host_reboot

Status:

Believed to be stable

Function — Reboot this host

SYNOPSIS

kern_return_t **host_reboot**
(host_control_port_t *host_priv,*
reboot_t *options*);

DESCRIPTION

The **host_reboot** function reboots the specified host.

PARAMETERS

host_priv
[in host-control send right] The control port the host to be re-booted.

options
[in scalar] Reboot options. See **<sys/reboot.h>** for details.

NOTES

If successful, this call will not return.

RETURN VALUE

Only generic errors apply.

Status:

Believed to be stable

host_statistics

Function — Returns statistics for a host

SYNOPSIS

kern_return_t **host_statistics**

(host_control_port_t	*host_priv,*
host_flavor_t	*flavor,*
host_info_t	*host_info*,
mach_msg_type_number_t*	*host_info_count*);

DESCRIPTION

The **host_statistics** function returns scheduling and virtual memory statistics concerning the host as specified by *flavor.*

PARAMETERS

host_priv
[in host-control send right] The control port for the host for which information is to be obtained.

flavor
[in scalar] The type of statistics desired.

HOST_LOAD_INFO
System loading statistics. The returned structure is **host_load_info**.

HOST_VM_INFO
Virtual memory statistics. The returned structure is **vm_statistics**.

host_info
[out structure] Statistics about the specified host.

host_info_count
[in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info**, **processor_set_statistics**.

Data Structures: **host_load_info**, **vm_statistics**.

Status:

Believed to be stable

mach_host_self

System Trap — Returns the host self port

SYNOPSIS

host_name_port_t **mach_host_self**
();

DESCRIPTION

The **mach_host_self** function returns send rights to the task's host self port. By default, this is the name port for the current host but can be a different value if so set.

PARAMETERS

None

RETURN VALUE

[host-self send right] Send rights to the host's name port.

RELATED INFORMATION

Functions: **host_info**, **task_set_special_port**.

CHAPTER 9 Processor Management and Scheduling Interface

This chapter discusses the specifics of the kernel's processor and processor set interfaces. This includes functions to control processors, change their assignments, assign tasks and threads to processors, and processor status returning functions.

Note that processor sets have two ports that name them: a name port which allows information to be requested about them, and a control port which allows full access. The control port for a processor set is provided to the creator of the set.

Processors have only a single port that names them. The host control port is needed to obtain these processor ports.

Status:

Believed to be stable

host_processor_set_priv

Function — Translates a processor set name port into a processor set control port

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t host_processor_set_priv
                (host_control_port_t                    host_priv,
                 processor_set_name_port_t              set_name,
                 processor_set_control_port_t*          processor_set);
```

DESCRIPTION

The **host_processor_set_priv** function returns send rights for the control port for a specified processor set currently existing on *host_priv*.

PARAMETERS

host_priv
: [in host-control send right] The control port for the host for which the processor set is desired.

set_name
: [in processor-set-name send right] The name port for the processor set desired.

processor_set
: [out processor-set-control send right] The returned processor set control port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_processor_sets, processor_set_create, processor_set_tasks, processor_set_threads**.

host_processor_sets

Status:

Believed to be stable

Function — Returns processor set ports for a host

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **host_processor_sets**

(host_control_port_t	*host_priv,*
processor_set_name_port_array_t*	*processor_set_name_list,*
mach_msg_type_number_t*	*processor_set_name_count*);

DESCRIPTION

The **host_processor_sets** function returns send rights for the name ports for each processor set currently existing on *host*.

PARAMETERS

host_priv
[in host-control send right] The control port for the host for which the processor sets are desired.

processor_set_name_list
[out pointer to dynamic array of processor-set-name send rights] The set of processor set name ports for those currently existing on *host_priv*; no particular order is guaranteed.

processor_set_name_count
[out scalar] The number of processor set names returned.

NOTES

If control ports to the processor sets are needed, use **host_processor_set_priv**.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_processor_set_priv, processor_set_create, processor_set_tasks, processor_set_threads**.

Status:

Believed to be stable

host_processor_slots

Function — Returns the defined processor slots

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

kern_return_t **host_processor_slots**

(host_name_port_t	*host,*
processor_slot_t*	*slots,*
mach_msg_type_number_t*	*count*);

DESCRIPTION

The **host_processor_slots** function returns an array of the processor slots defined for the host.

PARAMETERS

host

[in host-name send right] The name (or control) port for the host for which information is to be obtained.

slots

[out array of *processor_slot_t*] An array of the processor slot numbers for active processors.

count

[pointer to in/out scalar] On input, the maximum size of the *slots* buffer; on output, the number of returned slot numbers.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info, host_processors, processor_info**.

host_processors

Status:

Believed to be stable

Function — Gets processor ports for a host

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **host_processors**
(host_control_port_t *host_priv,*
processor_port_array_t* *processor_list,*
mach_msg_type_number_t* *processor_count*);

DESCRIPTION

The **host_processors** function returns an array of send right ports for each processor existing on *host_priv*.

PARAMETERS

host_priv
[in host-control send right] The control port for the desired host.

processor_list
[out pointer to dynamic array of processor send rights] The set of processors existing on *host_priv*; no particular order is guaranteed.

processor_count
[out scalar] The number of ports returned in *processor_list*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **processor_start, processor_exit, processor_info, processor_control**, **host_processor_slots**.

Status:

Believed to be stable

processor_assign

Function — Assign a processor to a processor set

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t processor_assign
        (processor_port_t                processor,
         processor_set_control_port_t    new_set,
         boolean_t                       wait);
```

DESCRIPTION

The **processor_assign** function assigns *processor* to the set *new_set*. After the assignment is completed, the processor only executes threads that are assigned to that processor set. Any previous assignment of the processor is nullified. The master processor cannot be reassigned.

The *wait* argument indicates whether the caller should wait for the assignment to be completed or should return immediately. Dedicated kernel threads are used to perform processor assignment, so setting *wait* to FALSE allows assignment requests to be queued and performed quicker, especially if the kernel has more than one dedicated internal thread for processor assignment.

All processors take clock interrupts at all times. Redirection of other device interrupts away from processors assigned to other than the default processor set is machine dependent.

PARAMETERS

processor
: [in processor send right] The processor to be assigned.

new_set
: [in processor-set-control send right] The control port for the processor set into which the processor is to be assigned.

wait
: [in scalar] True if the call should wait for the completion of the assignment.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_create, processor_set_info, task_assign**, **thread_assign**.

processor_control

Status:

Believed to be stable

Function — Do something to a processor

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t processor_control
        (processor_port_t                processor,
         processor_info_t                cmd,
         mach_msg_type_number_t          count);
```

DESCRIPTION

The **processor_control** function allows privileged software to control a processor in a multi-processor that so allows it. The interpretation of *cmd* is machine dependent.

PARAMETERS

processor
: [in processor send right] The processor to be controlled.

cmd
: [pointer to in array of natural-sized units] An array containing the command to be applied to the processor.

count
: [in scalar] The size of the *cmd* array (in natural-sized units).

NOTES

These operations are machine dependent. They may do nothing.

RETURN VALUE

KERN_FAILURE
: The operation was not performed. A likely reason is that it is not supported on this processor.

RELATED INFORMATION

Functions**: processor_start, processor_exit, processor_info, host_processors**.

processor_exit

Status:

Believed to be stable

Function — Exit a processor

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t processor_exit
                (processor_port_t                    processor);
```

DESCRIPTION

The **processor_exit** function allows privileged software to exit a processor in a multi-processor that so allows it. An exited processor is removed from the processor set to which it was assigned and ceases to be active. The interpretation of this operation is machine dependent.

PARAMETERS

processor
: [in processor send right] The processor to be controlled.

NOTES

This operation is machine dependent. It may do nothing.

CAUTIONS

The ability to restart an exited processor is machine dependent.

RETURN VALUE

KERN_FAILURE
: The operation was not performed. A likely reason is that it is not supported on this processor.

RELATED INFORMATION

Functions: **processor_control, processor_start, processor_info, host_processors**.

Status:

Believed to be stable

processor_get_assignment

Function — Get current assignment for a processor

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

kern_return_t **processor_get_assignment**
    (processor_port_t   *processor,*
    processor_set_name_port_t*   *assigned_set*);

DESCRIPTION

The **processor_get_assignment** function returns the name port for the processor set to which a desired processor is currently assigned.

PARAMETERS

processor
: [in processor send right] The processor whose assignment is desired.

new_set
: [out processor-set-name send right] The name port for the processor set to which *processor* is currently assigned.

RETURN VALUE

KERN_FAILURE
: *processor* is either shut down of off-line.

RELATED INFORMATION

Functions: **processor_assign, processor_set_create, processor_info, task_assign**, **thread_assign**.

processor_info

Status:

Believed to be stable

Function — Returns information about a processor.

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t processor_info
                (processor_port_t                         processor,
                processor_flavor_t                        flavor,
                host_name_port_t*                         host,
                processor_info_t                          processor_info,
                mach_msg_type_number_t*                   processor_info_count);
```

DESCRIPTION

The **processor_info** function returns selected information for a processor, as specified by *flavor*.

PARAMETERS

processor
: [in processor send right] A processor port for which information is desired.

flavor
: [in scalar] The type of information requested.

  PROCESSOR_BASIC_INFO
  : Basic information, slot number, running status, etc. The returned structure is **processor_basic_info**.

host
: [out host-name send right] The host on which the processor resides. This is the host name port.

processor_info
: [out structure] Information about the processor.

processor_info_count
: [in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **processor_start, processor_exit, processor_control, host_processors**.

Data Structures: **processor_basic_info.**

processor_set_create

Status:

Believed to be stable except for ledger reference

Function — Creates a new processor set

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t processor_set_create
        (host_name_port_t                    host_name,
         processor_set_control_port_t*         new_set,
         processor_set_name_port_t*           new_name);
```

DESCRIPTION

The **processor_set_create** function creates a new processor set and returns the two ports associated with it. The port returned in *new_set* is the control port representing the set. It is used to perform operations such as assigning processors, tasks or threads. The port returned in *new_name* is the name port which identifies the set, and is used to obtain information about the set.

PARAMETERS

host_name
: [in host-name send right] The name (or control) port for the host on which the set is to be created.

new_set
: [out processor-set-control send right] Control port used for performing operations on the new set.

new_name
: [out processor-set-name send right] Name port used to identify the new set and obtain information about it.

RETURN VALUE

KERN_INVALID_LEDGER
: *ledger* is not a wired kernel memory ledger.

RELATED INFORMATION

Functions**: processor_set_destroy, processor_set_info, processor_assign, task_assign**, **thread_assign**.

processor_set_default

Status:

Believed to be stable

Function — Returns the default processor set

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

kern_return_t **processor_set_default**
(host_name_port_t *host*,
processor_set_name_port_t* *default_set_name*);

DESCRIPTION

The **processor_set_default** function returns the name port for the default processor set for the specified host. The default processor set is used by all threads, tasks and processors that are not explicitly assigned to other sets.

PARAMETERS

host
: [in host-name send right] The name (or control) port for the host for which the default processor set is desired.

default_set_name
: [out processor-set-name send right] The returned name port for the default processor set.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_info, thread_assign, task_assign**.

processor_set_destroy

Function — Destroys a processor set

Status:

Believed to be stable

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **processor_set_destroy**
(processor_set_control_port_t *processor_set*);

DESCRIPTION

The **processor_set_destroy** function destroys the specified processor set. Any assigned processors, tasks or threads are re-assigned to the default set. The object port (not the name port) for the processor set is required.

PARAMETERS

processor_set
[in processor-set-control send right] The control port for the processor set to be destroyed.

RETURN VALUE

KERN_DEFAULT_SET
An attempt was made to destroy the default processor set.

RELATED INFORMATION

Functions**: processor_set_create, processor_assign, task_assign**, **thread_assign**.

Status:

Believed to be stable

processor_set_info

Function — Returns information about a processor set.

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

kern_return_t **processor_set_info**

(processor_set_name_port_t	*processor_set_name,*
processor_set_flavor_t	*flavor,*
host_name_port_t*	*host,*
processor_set_info_t	*processor_set_info*,
mach_msg_type_number_t*	*processor_set_info_count*);

DESCRIPTION

The **processor_set_info** function returns selected information for a processor set, as specified by *flavor*.

PARAMETERS

processor_set_name
[in processor-set-name send right] A processor set name (or control) port for which information is desired.

flavor
[in scalar] The type of information requested.

PROCESSOR_SET_BASIC_INFO
Basic information concerning the processor set (number of assigned processors and default policy). The returned structure is defined by **processor_set_basic_info**.

PROCESSOR_SET_TIMESHARE_DEFAULT
The base attributes for the timeshare scheduling policy. The returned structure is **policy_timeshare_base**.

PROCESSOR_SET_FIFO_DEFAULT
The base attributes for the FIFO scheduling policy. The returned structure is **policy_fifo_base**.

PROCESSOR_SET_RR_DEFAULT
The base attributes for the round-robin scheduling policy. The returned structure is **policy_rr_base**.

PROCESSOR_SET_TIMESHARE_LIMITS
Limits on the allowed timeshare policy attributes. The returned structure is defined by **policy_timeshare_limit**.

PROCESSOR_SET_RR_LIMITS
Limits on the allowed round robin policy attributes. The returned structure is defined by **policy_rr_limit**.

PROCESSOR_SET_FIFO_LIMITS
Limits on the allowed first-in, first-out policy attributes. The returned structure is defined by **policy_fifo_limit**.

PROCESSOR_SET_ENABLED_POLICIES
The set of enabled policies. The returned data is a bit-vector.

host
[out host-name send right] The name port for the host on which the processor set resides.

processor_set_info
[out structure] Information about the processor set.

processor_set_info_count
[in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

NOTES

A processor set has a single default scheduling policy in effect for it (as returned by PROCESSOR_SET_BASIC_INFO), so only one of the default scheduling structures has valid information. On the other hand, a processor set maintains limits for all defined scheduling policies, so all of the scheduling limit structures return valid values.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_statistics**, **processor_set_create, processor_set_default**, **processor_assign**, **processor_set_policy_control**.

Data Structures: **processor_set_basic_info, policy_timeshare_info, policy_rr_info, policy_fifo_info.**

Status:

Believed to be stable

processor_set_policy_control

Function — Controls the scheduling attributes governing the processor set

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

kern_return_t **processor_set_policy_control**

(processor_set_control_port_t	*processor_set_control,*
processor_set_flavor_t	*flavor,*
processor_set_info_t	*policy_info*,
mach_msg_type_number_t	*policy_info_count*,
boolean_t	*change_tasks_threads*);

DESCRIPTION

The **processor_set_policy_control** function controls scheduling attributes governing the processor set.

PARAMETERS

processor_set_control
: [in processor-set-control send right] A processor set control port.

flavor
: [in scalar] The type of policy change to make.

PROCESSOR_SET_TIMESHARE_DEFAULT
: Change the base attributes for the timeshare scheduling policy, making timeshare the default policy. The structure is **policy_timeshare_base**.

PROCESSOR_SET_FIFO_DEFAULT
: Change the base attributes for the FIFO (first-in, first-out) scheduling policy, making FIFO the default policy. The structure is **policy_fifo_base**.

PROCESSOR_SET_RR_DEFAULT
: Changed the base attributes for the round-robin scheduling policy, making round robin the default policy. The structure is **policy_rr_base**.

PROCESSOR_SET_TIMESHARE_LIMITS
: Change the limits on the allowed timeshare policy attributes. The structure is defined by **policy_timeshare_limit**.

PROCESSOR_SET_RR_LIMITS
: Change the limits on the allowed round robin policy attributes. The structure is defined by **policy_rr_limit**.

PROCESSOR_SET_FIFO_LIMITS
: Change the limits on the allowed first-in, first-out policy attributes. The structure is defined by **policy_fifo_limit**.

PROCESSOR_SET_ENABLED_POLICIES
: Change the set of enabled policies. The data is a bit-vector.

policy_info
: [in structure] The relevant policy information.

policy_info_count
: [in scalar] The size of the buffer (in natural-sized units).

change_tasks_threads
: [in scalar] If true, any assigned task or thread whose policy is no longer enabled or whose scheduling attributes exceed the current limits will have their limits adjusted or their policy set to the default as appropriate.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_statistics**, **processor_set_create, processor_set_default**, **processor_assign**, **processor_set_info**.

Data Structures: **policy_timeshare_info, policy_rr_info, policy_fifo_info.**

Status:

Believed to be stable

processor_set_statistics

Function — Returns scheduling statistics for a processor set.

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t processor_set_statistics
        (processor_set_control_port_t        processor_set_control,
         processor_set_flavor_t              flavor,
         processor_set_info_t                processor_set_info,
         mach_msg_type_number_t*             processor_set_info_count);
```

DESCRIPTION

The **processor_set_statistics** function returns statistics for a processor set as specified by *flavor*.

PARAMETERS

processor_set_control
: [in processor-set-control send right] A processor set control port for which information is desired.

flavor
: [in scalar] The type of information requested.

  PROCESSOR_SET_LOAD_INFO
  : Load statistics for the processor set. The returned structure is **processor_set_load_info**.

processor_set_info
: [out structure] Information about the processor set.

processor_set_info_count
: [in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_info**.

Data Structures: **processor_set_load_info.**

Status:

Believed to be stable

processor_set_tasks

Function — Returns a list of tasks assigned to a processor set

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t processor_set_tasks
                (processor_set_control_port_t           processor_set,
                 task_port_array_t*                         task_list,
                 mach_msg_type_number_t*                   task_count);
```

DESCRIPTION

The **processor_set_tasks** function returns send rights to the kernel ports for each task currently assigned to *processor_set*.

PARAMETERS

processor_set
: [in processor-set-control send right] A processor set control port for which information is desired.

task_list
: [out pointer to dynamic array of task send rights] The returned set of ports naming the tasks currently assigned to *processor_set*.

task_count
: [out scalar] The number of tasks returned in *task_list*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_threads, task_assign**, **thread_assign**.

processor_set_threads

Function — Returns a list of threads assigned to a processor set

Status:

Believed to be stable

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t processor_set_threads
                (processor_set_control_port_t          processor_set,
                 thread_port_array_t*                  thread_list,
                 mach_msg_type_number_t*               thread_count);
```

DESCRIPTION

The **processor_set_threads** function returns send rights to the kernel ports for each thread currently assigned to *processor_set*.

PARAMETERS

processor_set
: [in processor-set-control send right] A processor set control port for which information is desired.

thread_list
: [out pointer to dynamic array of thread send rights] The returned set of ports naming the threads currently assigned to *processor_set*.

thread_count
: [out scalar] The number of threads returned in *thread_list*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_tasks, task_assign**, **thread_assign**.

Status:

Believed to be stable

processor_start

Function — Start a processor

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t processor_start
        (processor_port_t                    processor);
```

DESCRIPTION

The **processor_start** function allows privileged software to start a processor in a multi-processor that so allows it. A newly started processor is assigned to the default processor set. The interpretation of this operation is machine dependent.

PARAMETERS

processor
: [in processor send right] The processor to be controlled.

NOTES

This operation is machine dependent. It may do nothing.

CAUTIONS

The ability to restart an exited processor is machine dependent.

RETURN VALUE

KERN_FAILURE
: The operation was not performed. A likely reason is that it is not supported on this processor.

RELATED INFORMATION

Functions**: processor_control, processor_exit, processor_info, host_processors**.

task_assign

Status:

Believed to be stable

Function — Assign a task to a processor set

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t task_assign
                (task_port_t                          task,
                 processor_set_control_port_t         processor_set,
                 boolean_t                            assign_threads);
```

DESCRIPTION

The **task_assign** function assigns *task* to the set *processor_set*. After the assignment is completed, newly created threads within this task will be assigned to this processor set. Any previous assignment of the task is nullified.

If *assign_threads* is TRUE, existing threads within the task will also be assigned to the processor set.

PARAMETERS

task
: [in task send right] The port for the task to be assigned.

processor_set
: [in processor-set-control send right] The control port for the processor set into which the task is to be assigned.

assign_threads
: [in scalar] True if this assignment should apply as well to the threads within the task.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: task_assign_default, task_get_assignment, processor_set_create, processor_set_info**, **thread_assign**.

Status:

Believed to be stable

task_assign_default

Function — Assign a task to the default processor set

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t task_assign_default
        (task_port_t                    task,
         boolean_t                      assign_threads);
```

DESCRIPTION

The **task_assign_default** function assigns *task* to the default processor set. After the assignment is completed, newly created threads within this task will be assigned to this processor set. Any previous assignment of the task is nullified.

If *assign_threads* is TRUE, existing threads within the task will also be assigned to the processor set.

PARAMETERS

task
: [in task send right] The port for the task to be assigned.

assign_threads
: [in scalar] True if this assignment should apply as well to the threads within the task.

NOTES

This variant of **task_assign** exists because the control port for the default processor set is privileged, and therefore not available to most tasks.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: task_assign, task_get_assignment, processor_set_create, processor_set_info, thread_assign**.

task_get_assignment

Status:

Believed to be stable

Function — Returns the processor set to which a task is assigned

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t task_get_assignment
                (task_port_t                         task,
                 processor_set_name_port_t*          processor_set);
```

DESCRIPTION

The **task_get_assignment** function returns the name port to the processor set to which *task* is currently assigned. This port can only be used to obtain information about the processor set.

PARAMETERS

task
: [in task send right] The port for the task whose assignment is desired.

processor_set
: [out processor-set-name send right] The name port for the processor set into which the task is assigned.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: task_assign, task_assign_default, processor_set_create, processor_set_info, thread_assign**.

Status:

Believed to be stable

task_policy

Function — Sets the default scheduling attributes for a task

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t task_policy
        (task_port_t                    task,
         policy_t                       policy,
         policy_base_t                  base,
         mach_msg_type_number_t         base_count,
         boolean_t                      set_limit,
         boolean_t                      change_threads);
```

DESCRIPTION

The **task_policy** function sets the default scheduling attributes for *task*. These attributes are used when creating new threads. Changing the default attributes for a task does not affect the attributes of the contained threads unless *change_threads* is TRUE. At no time will a thread ever have scheduling attributes that exceed the thread's limits.

PARAMETERS

task
: [in task send right] The port for the task whose scheduling attributes are to be set.

policy
: [in scalar] Default policy. The values currently defined are POLICY_-TIMESHARE, POLICY_RR (round robin) and POLICY_FIFO (first-in, first-out).

base
: [pointer to in structure] Base scheduling policy data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

base_count
: [in scalar] The size of the buffer (in natural-sized units).

set_limit
: [in scalar] True if the scheduling limits for the task should be restricted to allow no more service than specified by *base*.

change_threads
[in scalar] True if the attributes (and limits, if *set_limit* is true) of existing threads within the task should also be changed.

RETURN VALUE

KERN_INVALID_POLICY
The processor set does not currently enable *policy*.

KERN_POLICY_LIMIT
The specified scheduling attributes exceeds the thread's limits.

RELATED INFORMATION

Functions: **thread_policy**, **thread_set_policy**, **task_set_policy**, **processor_set_policy_control**.

Data structures: **policy_fifo_info**, **policy_rr_info**, **policy_timeshare_info**.

Status:

Believed to be stable

task_set_policy

Function — Sets the scheduling policy to apply to a task

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

kern_return_t **task_set_policy**

(task_port_t	*task,*
processor_set_control_port_t	*processor_set,*
policy_t	*policy,*
policy_base_t	*base*,
mach_msg_type_number_t	*base_count*,
policy_limit_t	*limit*,
mach_msg_type_number_t	*limit_count*,
boolean_t	*change_threads*);

DESCRIPTION

The **task_set_policy** function sets the scheduling attributes, both base and limit, for *task*. *policy* may be any policy implemented by the processor set whether or not it is enabled.

PARAMETERS

task
: [in task send right] The task whose scheduling policy is to be set.

processor_set
: [in processor-set-control send right] The control port for the processor set to which the task is currently assigned.

policy
: [in scalar] Policy to be set. The values currently defined are POLICY_-TIMESHARE, POLICY_RR (round robin) and POLICY_FIFO (first-in, first-out).

base
: [pointer to in structure] Base policy specific data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

base_count
: [in scalar] The size of the buffer (in natural-sized units).

limit

[pointer to in structure] Policy specific limits, **policy_fifo_limit**, **policy_rr_limit** or **policy_timeshare_limit**.

limit_count

[in scalar] The size of the buffer (in natural-sized units).

change_threads

[in scalar] True if the scheduling attributes for all contained threads should be changed as well.

RETURN VALUE

KERN_INVALID_PROCESSOR_SET

processor_set is not the task's processor set control port.

RELATED INFORMATION

Functions: **processor_set_policy_control**, **thread_policy**, **thread_set_policy**, **task_policy**.

Data structures: **policy_fifo_info**, **policy_rr_info**, **policy_timeshare_info**.

Status:

Believed to be stable

thread_assign

Function — Assign a thread to a processor set

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **thread_assign**
(thread_port_t *thread,*
processor_set_control_port_t *processor_set*);

DESCRIPTION

The **thread_assign** function assigns *thread* to the set *processor_set*. After the assignment is completed, the thread executes only on processors that are assigned to that processor set. Any previous assignment of the thread is nullified.

PARAMETERS

thread
[in thread send right] The thread to be assigned.

processor_set
[in processor-set-control send right] The control port for the processor set into which the thread is to be assigned.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: thread_assign_default, thread_get_assignment, processor_set_create, processor_set_info, task_assign**.

thread_assign_default

Function — Assign a thread to the default processor set

Status:

Believed to be stable

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **thread_assign_default**
(thread_port_t *thread*);

DESCRIPTION

The **thread_assign_default** function assigns *thread* to the default processor set. After the assignment is completed, the thread executes only on processors that are assigned to that processor set. Any previous assignment of the thread is nullified.

PARAMETERS

thread
[in thread send right] The thread to be assigned.

NOTES

This variant of **thread_assign** exists because the control port for the default processor set is privileged, and therefore not available to most tasks.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: thread_assign, thread_get_assignment, processor_set_create, processor_set_info, task_assign**.

Status:

Believed to be stable

thread_get_assignment

Function — Returns the processor set to which a thread is assigned

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t thread_get_assignment
        (thread_port_t              thread,
         processor_set_name_t*      processor_set);
```

DESCRIPTION

The **thread_get_assignment** function returns the name port to the processor set to which *thread* is currently assigned. This port can only be used to obtain information about the processor set.

PARAMETERS

thread
: [in thread send right] The thread whose assignment is desired.

processor_set
: [out processor-set-name send right] The name port for the processor set into which the thread is assigned.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: thread_assign, thread_assign_default, processor_set_create, processor_set_info, task_assign**.

thread_policy

Status:

Believed to be stable

Function — Sets the scheduling policy to apply to a thread

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t thread_policy
        (thread_port_t                 thread,
         policy_t                      policy,
         policy_base_t                 base,
         mach_msg_type_number_t        base_count,
         boolean_t                     set_limit);
```

DESCRIPTION

The **thread_policy** function sets the scheduling policy to be applied to *thread*. *policy* must be a scheduling policy currently "enabled" for the thread's assigned processor set.

PARAMETERS

thread
: [in thread send right] The thread scheduling policy is to be set.

policy
: [in scalar] Policy to be set. The values currently defined are POLICY_TIMESHARE, POLICY_RR (round robin) and POLICY_FIFO (first-in, first-out).

base
: [pointer to in structure] Base scheduling policy specific data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

base_count
: [in scalar] The size of the buffer (in natural-sized units).

set_limit
: [in scalar] True if the thread's scheduling limits should be restricted to allow no more service than specified by *base*.

RETURN VALUE

KERN_INVALID_POLICY
The processor set to which *thread* is currently assigned does not currently enable *policy*.

KERN_POLICY_LIMIT
The specified scheduling attributes exceeds the thread's limits.

RELATED INFORMATION

Functions: **processor_set_policy_control**, **thread_set_policy**, **task_policy**, **task_set_policy**.

Data structures: **policy_fifo_info**, **policy_rr_info**, **policy_timeshare_info**.

thread_set_policy

Status:

Believed to be stable

Function — Sets the scheduling policy to apply to a thread

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **thread_set_policy**

(thread_port_t	*thread,*
processor_set_control_port_t	*processor_set,*
policy_t	*policy,*
policy_base_t	*base*,
mach_msg_type_number_t	*base_count*,
policy_limit_t	*limit*,
mach_msg_type_number_t	*limit_count*);

DESCRIPTION

The **thread_set_policy** function sets the scheduling attributes, both base and limit, for *thread*. *policy* may be any policy implemented by the processor set whether or not it is enabled.

PARAMETERS

thread
: [in thread send right] The thread scheduling policy is to be set.

processor_set
: [in processor-set-control send right] The control port for the processor set to which the thread is currently assigned.

policy
: [in scalar] Policy to be set. The values currently defined are POLICY_-TIMESHARE, POLICY_RR (round robin) and POLICY_FIFO (first-in, first-out).

base
: [pointer to in structure] Base policy specific data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

base_count
: [in scalar] The size of the buffer (in natural-sized units).

limit
: [pointer to in structure] Policy specific limits, **policy_fifo_limit**, **policy_rr_limit** or **policy_timeshare_limit**.

limit_count
: [in scalar] The size of the buffer (in natural-sized units).

RETURN VALUE

KERN_INVALID_PROCESSOR_SET
: *processor_set* is not the thread's processor set control port.

RELATED INFORMATION

Functions: **processor_set_policy_control**, **thread_policy**, **task_policy**, **task_set_policy**.

Data structures: **policy_fifo_info**, **policy_rr_info**, **policy_timeshare_info**.

CHAPTER 10 In-Kernel Device Interface

This chapter discusses the specifics of the device interfaces to in-kernel device drivers. These interfaces provide read, write and status interfaces to devices.

A new set of interfaces have been added to provide for asynchronous I/O support which allows for completion notification directly from the device driver interrupt context.

Status:

Believed to be stable

device_close

Function — De-establish a connection to a device.

LIBRARY

#include <**device/device.h**>

SYNOPSIS

kern_return_t **device_close**
(device_port_t *device*);

DESCRIPTION

The **device_close** function destroys the associated device port. The open count for the named device is decremented. If this count reaches zero, the close operation of the device driver is invoked, closing the device.

PARAMETERS

device
[in device send right] A device port to the device to be closed.

NOTES

device_close will destroy any mapped device windows obtained through this device port.

RETURN VALUE

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.

RELATED INFORMATION

Functions: **device_open**.

device_get_status

Status:

Believed to be stable

The MDIOCGETL flavor is experimental. **[OSF-RI DIPC]**

Function — Return the current device status

LIBRARY

#include **<device/device.h>**
#include **<device/ds_status.h>**

SYNOPSIS

```
kern_return_t device_get_status
        (device_port_t                  device,
         device_flavor_t                flavor,
         dev_status_t                   status,
         mach_msg_type_number_t*        status_count);
```

DESCRIPTION

The **device_get_status** function returns status information pertaining to an open device. The possible values for *flavor* as well as the meaning of the returned status information is device dependent, except for MDIOCGETL which is generic.

PARAMETERS

device
: [in device send right] A device port to the device to be interrogated.

flavor
: [in scalar] The type of status information requested. The only device generic flavor is:

  MDIOCGETL
  Returns the current value of the device's I/O operations limit (i.e. the number of outstanding requests on the device).

status
: [out array of natural-sized units] The returned device status.

status_count
: [pointer to in/out scalar] On input, the reserved size of *status*; on output, the size of the returned device status (in natural-sized units).

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.

D_OUT_OF_BAND
Out-of-band condition occurred on device (such as typing control-C)

RELATED INFORMATION

Functions: **device_set_status**.

device_map

Status:

Believed to be stable

Function — Establish a memory manager representing a device

LIBRARY

#include <**device/device.h**>

SYNOPSIS

kern_return_t **device_map**

(device_port_t	*device,*
vm_prot_t	*prot,*
vm_size_t	*size,*
mem_obj_rep_port_t*	*pager*);

DESCRIPTION

The **device_map** function establishes a memory manager that presents a memory object representing a device. The resulting port is suitable for use as the memory manager port in a **vm_map** call. This call is device dependent.

PARAMETERS

device
: [in device send right] A device port to the device to be mapped.

prot
: [in scalar] Protection for the device memory.

size
: [in scalar] The size of the device memory object.

pager
: [out memory-object-representative send right] The returned memory object representative port to a memory manager that represents the device.

NOTES

Port rights are maintained as follows:

Memory object representative port (and abstract memory object port):
: The device memory manager has all rights.

Memory cache control port:
: The device memory manager has only send rights.

Regardless how the object is created, the control port is created by the kernel and passed through the memory management interface.

If the device port used is restricted to use by a single identity, the generated representative port will be likewise restricted.

CAUTIONS

The device memory manager assumes that access to its memory objects will not be propagated to more that one host, and therefore provides no consistency guarantees beyond those made by the kernel.

In the event that more than one host attempts to use a device memory object, the device memory manager will only record the last set of port names. Currently, the device memory manager assumes that its clients adhere to the initialization and termination protocols in the memory management interface; otherwise, port rights or out-of-line memory from erroneous messages may be allowed to accumulate.

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

D_READ_ONLY
: Data cannot be written to this device.

RELATED INFORMATION

Functions: **vm_map**.

device_open

Function — Establish a connection to a device.

Status:

Believed to be stable

LIBRARY

#include <**device/device.h**> (**device_open**)

#include <**device/device_request.h**> (**device_open_request**)

#include <**device/device_reply.h**> (**ds_device_open_reply**)

SYNOPSIS

```
kern_return_t device_open
        (device_master_port_t        master_port,
         dev_mode_t                  mode,
         security_token_t            security_id,
         dev_name_t                  name,
         device_port_t*              device);
```

device_open_request

Asynchronous Function form — Asynchronously request a connection to a device

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

```
kern_return_t device_open_request
        (device_master_port_t        master_port,
         mach_port_t                 reply_port,
         dev_mode_t                  mode,
         security_token_t            security_token,
         dev_name_t                  name);
```

ds_device_open_reply

Asynchronous Server Interface form — Receive the reply from an asynchronous open

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

```
kern_return_t ds_device_open_reply
        (mach_port_t                 reply_port,
         kern_return_t               return_code,
         device_port_t               device);
```

DESCRIPTION

The **device_open** function opens a device object. The open operation of the device is invoked, if the device is not already open. The open count for the device is incremented. Each open for a device returns a distinct port, the allowed operations upon which being governed by *mode*.

PARAMETERS

master_port
: [in device-master send right] The master device port. This port is provided to the bootstrap task.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which a reply is to be sent when the device is open.

mode
: [in scalar] Opening mode. This is the bit-wise OR of the following values:

  D_READ
  : Read access

  D_WRITE
  : Write access

  D_NODELAY
  : Do not delay on open

security_id
: [in scalar] The security ID that tasks attempting to use this device port must have. A zero value indicates all identities.

name
: [pointer to in array of *char*] Name of the device to open.

return_code
: [in scalar] Status of the open.

device
: [out device send right, in for asynchronous form] The returned device port.

RETURN VALUE

device_open_request returns only message transmission errors. The return value supplied to **ds_device_open_reply** is irrelevant. The *return_code* returned by **ds_device_open_reply** or the error return from **device_open** is one of the following:

D_WOULD_BLOCK
: The device is busy, but D_NOWAIT was specified in *mode*.

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

D_DEVICE_DOWN
The device has been shut down.

D_READ_ONLY
Data cannot be written to this device.

RELATED INFORMATION

Functions: **device_close**, **device_reply_server**.

Status:

Believed to be stable

Use of the I/O operations limit is experimental.
[OSF-RI DIPC]

device_read

Function — Read a sequence of bytes from a device object.

LIBRARY

#include <**device/device.h**> (**device_read**)

#include <**device/device_request.h**> (**device_read_request**)

#include <**device/device_reply.h**> (**ds_device_read_reply**)

SYNOPSIS

kern_return_t **device_read**

(device_port_t	*device,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted,*
kern_return_t*	*device_code*,
pointer_t*	*data,*
mach_msg_type_number_t*	*data_count*);

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

device_read_request

Asynchronous Function form — Asynchronously read data

kern_return_t **device_read_request**

(device_port_t	*device,*
mach_port_t	*reply_port,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted*);

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

ds_device_read_reply

Asynchronous Server Interface form — Receive the reply from an asynchronous read

kern_return_t **ds_device_read_reply**

(mach_port_t	*reply_port,*
kern_return_t	*return_code,*
kern_return_t	*device_code*,
pointer_t	*data,*
mach_msg_type_number_t	*data_count*);

DESCRIPTION

The **device_read** function reads a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device dependent.

The **device_read** function will block until the current number of operations on the device is less than the device's I/O operations limit.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which the reply message is to be sent.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
: [in scalar] Size of data transfer.

return_code
: [in scalar] The return status code from the read.

device_code
: [out scalar, in for asynchronous form] Device completion status. If the return value from the call (*return_code* in the asynchronous case) is not KERN_SUCCESS, the operation failed outright. If the operation partially succeeded, the return value is KERN_SUCCESS and *device_-code* reflects the status of the partial completion, D_SUCCESS if the operation was completely successful.

  D_DEVICE_DOWN
  : Device has been shut down

  D_INVALID_RECNUM
  : Invalid record (block) number

  D_INVALID_SIZE
  : Invalid IO size

  D_IO_ERROR
  : Hardware IO error

  D_NO_SUCH_DEVICE
  : No device with that name, or the device is not operational.

D_OUT_OF_BAND
Out-of-band condition occurred on device (such as typing control-C)

D_WOULD_BLOCK
Operation would block, but D_NOWAIT set

data
[out pointer to dynamic array of bytes, in for asynchronous form] Returned data bytes.

data_count
[out scalar, in for asynchronous form] Number of returned data bytes.

RETURN VALUE

device_read_request returns only message transmission errors. A return value supplied to **ds_device_read_reply** other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to de-allocate the returned data. The return value from **device_read** (or *return_code*) can be a device error code if that error caused the operation to fail completely.

RELATED INFORMATION

Functions: **device_read_inband**, **device_read_overwrite**, **device_reply_server**.

device_read_async

Function — Read a sequence of bytes from a device object.

Status:

New: [OSF-RI MK6] Added to support asynchronous I/O completion from interrupt context.

LIBRARY

#include **<device/device.h>** (**device_read_async**)

SYNOPSIS

kern_return_t **device_read_async**

(device_port_t	*device,*
mach_port_t	*queue;*
mach_port_t	*request_id;*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted)*;

DESCRIPTION

The **device_read_async** function enqueues a read operation for a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device dependent.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from **io_done_-queue_create**.

request_id
: [in send right] An unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_read_async returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**, **io_done_queue_create**.

device_read_inband

Function — Read a sequence of bytes “inband” from a device object.

Status:

Believed to be stable

Use of the I/O operations limit is experimental.
[OSF-RI DIPC]

LIBRARY

#include **<device/device.h>** (**device_read_inband**)

#include **<device/device_request.h>** (**device_read_request_inband**)

#include **<device/device_reply.h>** (**ds_device_read_reply_inband**)

SYNOPSIS

```
kern_return_t device_read_inband
        (device_port_t               device,
        dev_mode_t                   mode,
        recnum_t                     recnum,
        io_buf_len_t                 bytes_wanted,
        kern_return_t*               device_code,
        pointer_t                    data,
        mach_msg_type_number_t*      data_count);
```

device_read_request_inband

Asynchronous Function form — Asynchronously read data

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

```
kern_return_t device_read_request_inband
        (device_port_t               device,
        mach_port_t                  reply_port,
        dev_mode_t                   mode,
        recnum_t                     recnum,
        io_buf_len_t                 bytes_wanted);
```

ds_device_read_reply_inband

Asynchronous Server Interface form — Receive the reply from an asynchronous read

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

```
kern_return_t ds_device_read_reply_inband
        (mach_port_t                 reply_port,
        kern_return_t                return_code,
        kern_return_t                device_code,
        pointer_t                    data,
        mach_msg_type_number_t       data_count);
```

DESCRIPTION

The **device_read** function reads a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device dependent. This call differs from **device_read** in that the returned bytes are returned “inband” in the reply IPC message.

The **device_read_inband** function will block until the current number of operations on the device is less than the device's I/O operations limit.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which the reply message is to be sent.

mode
: [in scalar] I/O mode value. Meaningful options are:

    D_NOWAIT
    : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
: [in scalar] Size of data transfer.

return_code
: [in scalar] The return status code from the read.

device_code
: [out scalar, in for asynchronous form] Device completion status. If the return value from the call (*return_code* in the asynchronous case) is not KERN_SUCCESS, the operation failed outright. If the operation partially succeeded, the return value is KERN_SUCCESS and *device_-code* reflects the status of the partial completion, D_SUCCESS if the operation was completely successful.

    D_DEVICE_DOWN
    : Device has been shut down

    D_INVALID_RECNUM
    : Invalid record (block) number

    D_INVALID_SIZE
    : Invalid IO size

    D_IO_ERROR
    : Hardware IO error

    D_NO_SUCH_DEVICE
    : No device with that name, or the device is not operational.

D_OUT_OF_BAND
Out-of-band condition occurred on device (such as typing control-C)

D_WOULD_BLOCK
Operation would block, but D_NOWAIT set

data
[out array of bytes, in for asynchronous form] Returned data bytes.

data_count
[out scalar, in for asynchronous form] Number of returned data bytes.

RETURN VALUE

device_read_request_inband returns only message transmission errors. The return value supplied to **ds_device_read_reply_inband** is irrelevant. The return value from **device_read_inband** (or *return_code*) can be a device error code if that error caused the operation to fail completely.

RELATED INFORMATION

Functions: **device_read**, **device_read_overwrite, device_reply_server**.

Status:

New: [OSF-RI MK6] Added to support asynchronous I/O completion from interrupt context.

device_read_async_inband

Function — Read a sequence of bytes “inband” from a device object.

LIBRARY

#include <**device/device.h**> (**device_read_async_inband**)

SYNOPSIS

```
kern_return_t device_read_async_inband
        (device_port_t          device,
         mach_port_t            queue;
         mach_port_t            request_id;
         dev_mode_t             mode,
         recnum_t               recnum,
         io_buf_len_t           bytes_wanted);
```

DESCRIPTION

The **device_read_async_inband** function enqueues a read operation for a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device dependent. This call differs from **device_read_async** in that the returned bytes are returned “inband” in the completion IPC message (in io_done_result.qd_inline).

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from **io_done_-queue_create**.

request_id
: [in send right] An unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
[in scalar] Size of data transfer.

RETURN VALUE

device_read_async_inband returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async, device_read_overwrite_async, device_write_async, device_write_async_inband, io_done_queue_create**.

Status:

Believed to be stable

Use of the I/O operations limit is experimental.
[OSF-RI DIPC]

device_read_overwrite

System Trap — Read a sequence of bytes from a device object into the caller's address space

LIBRARY

#include **<device/device.h>** (**device_read_overwrite**)

#include **<device/device_request.h>** (**device_read_request_overwrite**)

#include **<device/device_reply.h>** (**ds_device_read_reply_overwrite**)

SYNOPSIS

kern_return_t **device_read_overwrite**

(device_port_t	*device,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted,*
pointer_t	*data,*
kern_return_t*	*device_code*,
io_buf_len_t*	*data_count*);

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

device_read_overwrite_request

Asynchronous System Trap form — Asynchronously read data into the caller's buffer

kern_return_t **device_read_overwrite_request**

(device_port_t	*device,*
mach_port_t	*reply_port,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted*,
pointer_t	*data*);

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

ds_device_read_reply_overwrite

Asynchronous Server Interface form — Receive the reply from an asynchronous read

kern_return_t **ds_device_read_reply_overwrite**

(mach_port_t	*reply_port,*
kern_return_t	*return_code,*
kern_return_t	*device_code*,
io_buf_len_t	*data_count*);

DESCRIPTION

The **device_read_overwrite** system trap reads a sequence of bytes from a device object directly into the caller's address space. The meaning of *recnum* as well as the specific operation performed is device dependent.

The **device_read_overwrite** function will block until the current number of operations on the device is less than the device's I/O operations limit.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which the reply message is to be sent.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
: [in scalar] Size of data transfer.

return_code
: [in scalar] The return status code from the read.

data
: [pointer to in array of bytes] Data buffer to be overwritten.

device_code
: [out scalar, in for asynchronous form] Device completion status. If the return value from the call (*return_code* in the asynchronous case) is not KERN_SUCCESS, the operation failed outright. If the operation partially succeeded, the return value is KERN_SUCCESS and *device_-code* reflects the status of the partial completion, D_SUCCESS if the operation was completely successful.

  D_DEVICE_DOWN
  : Device has been shut down

  D_INVALID_RECNUM
  : Invalid record (block) number

  D_INVALID_SIZE
  : Invalid IO size

  D_IO_ERROR
  : Hardware IO error

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

D_OUT_OF_BAND
: Out-of-band condition occurred on device (such as typing control-C)

D_WOULD_BLOCK
: Operation would block, but D_NOWAIT set

data_count
: [out scalar, in for asynchronous form] Number of returned data bytes.

RETURN VALUE

device_read_request_overwrite returns only invalid parameter errors. The return value supplied to **ds_device_read_reply_overwrite** is irrelevant. The return value from **device_read_overwrite** (or *return_code*) can be a device error code if that error caused the operation to fail completely.

RELATED INFORMATION

Functions: **device_read**, **device_read_inband**, **device_reply_server**.

device_read_overwrite_async

System Trap— Read a sequence of bytes from a device object into the caller's address space.

Status:

New: [OSF-RI MK6] Added to support asynchronous I/O completion from interrupt context.

LIBRARY

#include **<device/device.h>** (**device_read_overwrite_async**)

SYNOPSIS

kern_return_t **device_read_overwrite_async**

(device_port_t	*device,*
mach_port_t	*queue;*
mach_port_t	*request_id;*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted;*
io_buf_ptr_t	*buffer)*;

DESCRIPTION

The **device_read_overwrite** system trap enqueues a read operation for a sequence of bytes from a device object to be placed directly into the caller's address space. The meaning of *recnum* as well as the specific operation performed is device dependent.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from the **io_done_-queue_create** call.

request_id
: [in send right] An unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
[in scalar] Size of data transfer.

buffer
[pointer to in array of bytes] Data buffer to be overwritten.

RETURN VALUE

device_read_overwrite_async returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async**, **device_read_async_inband**, **device_write_async**, **device_write_async_inband**, **io_done_queue_create**.

device_set_filter

Status:

Believed to be stable

Function — Names an input filter for a device

LIBRARY

#include <**device/device.h**>

#include <**device/net_status.h**>

SYNOPSIS

kern_return_t **device_set_filter**

(device_port_t	*device,*
filter_port_t	*receive_port,*
mach_msg_type_name_t	*receive_port_type,*
net_priority_t	*priority,*
filter_array_t	*filter,*
mach_msg_type_number_t	*filter_count*);

DESCRIPTION

The **device_set_filter** function provides a means by which selected data appearing at a device interface can be selected and routed to a port. This service is provided in support of network devices.

The filter command list consists of an array of up to NET_MAX_FILTER (16-bit) values to be applied to incoming data “messages” to determine if that data should be given to a particular input filter.

Each filter command list specifies a sequence of actions which leave a boolean value on the top of an internal stack. Each 16-bit value of the command list specifies a data (push) operation (high order NETF_NBPO bits) as well as a binary operator (low order NETF_NBPA bits).

The value to be pushed onto the stack is chosen as follows.

NETF_PUSHLIT
: Use the next 16-bit value of the filter as the value.

NETF_PUSHZERO
: Use 0 as the value.

NETF_PUSHWORD+*N*
: Use 16-bit value *N* of the “data” portion of the message as the value.

NETF_PUSHHDR+*N*
: Use 16-bit value *N* of the “header” portion of the message as the value.

NETF_PUSHIND
: Pops the top 32-bit value from the stack and then uses it to index the 16-bit value of the “data” portion of the message to be used as the value.

NETF_PUSHHDRIND
: Pops the top 32-bit value from the stack and then uses it to index the 16-bit value of the “header” portion of the message to be used as the value.

NETF_PUSHSTK+*N*
: Use 32-bit value *N* of the stack (where the top of stack is value 0) as the value.

NETF_NOPUSH
: Don’t push a value.

The unsigned value so chosen is promoted to a 32-bit value before being pushed.

Once a value is pushed (except for the case of NETF_NOPUSH), the top two 32-bit values of the stack are popped and a binary operator applied to them (with the old top of stack as the second operand). The result of the operator is pushed on the stack. These operators are:

NETF_NOP
: Don’t pop off any values and do no operation.

NETF_EQ
: Perform an equal comparison.

NETF_LT
: Perform a less than comparison.

NETF_LE
: Perform a less than or equal comparison.

NETF_GT
: Perform a greater than comparison.

NETF_GE
: Perform a greater than or equal comparison.

NETF_AND
: Perform a bit-wise boolean AND operation.

NETF_OR
: Perform a bit-wise boolean inclusive OR operation.

NETF_XOR
: Perform a bit-wise boolean exclusive OR operation.

NETF_NEQ
: Perform a not equal comparison.

NETF_LSH
: Perform a left shift operation.

NETF_RSH
: Perform a right shift operation.

NETF_ADD
: Perform an addition.

NETF_SUB
: Perform a subtraction.

NETF_COR
: Perform an equal comparison. If the comparison is TRUE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CAND
: Perform an equal comparison. If the comparison is FALSE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CNOR
: Perform a not equal comparison. If the comparison is FALSE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CNAND
: Perform a not equal comparison. If the comparison is TRUE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

The scan of the filter list terminates when the filter list is emptied, or a NETF_C... operation terminates the list. At this time, if the final value of the top of the stack is TRUE, then the message is accepted for the filter.

PARAMETERS

device
: [in device send right] A device port

receive_port
: [in filter send or receive (to be converted to send) right] The port to receive the input data that is selected by the filter.

receive_port_type
: [in scalar] IPC type of the send right provided to the device; either MACH_MSG_TYPE_MAKE_SEND, MACH_MSG_TYPE_MOVE_-SEND or MACH_MSG_TYPE_COPY_SEND.

priority
: [in scalar] Used to order multiple receivers in an implementation dependent way.

filter
: [pointer to in array of *filter_t*] The address of an array of filter values.

filter_count
: [in scalar] The size of the *filter* array (in 16-bit values).

NOTES

The kernel implementation of a network device is free to drop packets. For example, a network device will drop a message (i.e. a packet) if the attempt to send the message encounters a full queue. This is necessary to avoid flooding receivers with packets.

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_INVALID_OPERATION
: No filter port was supplied.

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

RELATED INFORMATION

Functions: **receive_packet**.

device_set_status

Status:

Believed to be stable

The MDIOCSETL flavor is experimental. **[OSF-RI DIPC]**

Function — Sets device status.

LIBRARY

#include **<device/device.h>**
#include **<device/ds_status.h>**

SYNOPSIS

```
kern_return_t device_set_status
        (device_port_t              device,
         device_flavor_t            flavor,
         dev_status_t               status,
         mach_msg_type_number_t     status_count);
```

DESCRIPTION

The **device_set_status** function sets device status. The possible values of *flavor* as well as the corresponding meanings are device dependent, except for MDIOCSETL which is generic.

PARAMETERS

device
: [in device send right] A device port to the device to be manipulated.

flavor
: [in scalar] The type of status information to set. The only device generic flavor is:

  MDIOCSETL
  Returns the current value of the device's I/O operations limit (i.e. the number of outstanding requests on the device).The new value must be greater than or equal to ZERO and less than IO_IN_PROGRESS_MAX.

status
: [pointer to in array of natural-sized units] The status information to set.

status_count
: [in scalar] The size of the status information (in natural-sized units).

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_INVALID OPERATION
The proposed MDIOCSETL value was not in the range 0 through IO_-IN_PROGRESS_MAX.

D_IO_ERROR
Hardware IO error

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.

D_OUT_OF_BAND
Out-of-band condition occurred on device (such as typing control-C)

D_READ_ONLY
Data cannot be written to this device.

RELATED INFORMATION

Functions: **device_get_status**.

device_write

Status:

Believed to be stable

Use of the I/O operations limit is experimental.
[OSF-RI DIPC]

Function — Write a sequence of bytes to a device object.

LIBRARY

#include **<device/device.h>** (**device_write**)

#include **<device/device_request.h>** (**device_write_request**)

#include **<device/device_reply.h>** (**ds_device_write_reply**)

SYNOPSIS

kern_return_t **device_write**

(device_port_t	*device,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
pointer_t	*data,*
mach_msg_type_number_t	*data_count,*
boolean_t	*deallocate*,
kern_return_t*	*device_code*,
io_buf_len_t*	*bytes_written*);

device_write_request

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

Asynchronous Function form — Asynchronously write data

kern_return_t **device_write_request**

(device_port_t	*device,*
mach_port_t	*reply_port,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
pointer_t	*data,*
mach_msg_type_number_t	*data_count*,
boolean_t	*deallocate*);

ds_device_write_reply

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

Asynchronous Server Interface form — Receive the reply from an asynchronous write

kern_return_t **ds_device_write_reply**

(mach_port_t	*reply_port,*
kern_return_t	*return_code,*
kern_return_t	*device_code*,
io_buf_len_t	*bytes_written*);

DESCRIPTION

The **device_write** function writes a sequence of bytes to a device object. The meaning of *recnum* as well as the specific operation performed is device dependent.

The **device_write** function will block until the current number of operations on the device is less than the device's I/O operations limit.

PARAMETERS

device
: [in device send right] A device port to the device to be written.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which the reply message is to be sent.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait for I/O completion.

recnum
: [in scalar] Record number to be written.

data
: [pointer to in array of bytes] Data bytes to be written.

data_count
: [in scalar] Number of data bytes to be written.

deallocate
: [in scalar] If TRUE, the pages to be written will be de-allocated from the caller's address space.

return_code
: [in scalar] The return status code from the write.

device_code
: [out scalar, in for asynchronous form] Device completion status. If the return value from the call (*return_code* in the asynchronous case) is not KERN_SUCCESS, the operation failed outright. If the operation partially succeeded, the return value is KERN_SUCCESS and *device_-code* reflects the status of the partial completion, D_SUCCESS if the operation was completely successful.

  D_DEVICE_DOWN
  : Device has been shut down

  D_INVALID_RECNUM
  : Invalid record (block) number

D_INVALID_SIZE
: Invalid IO size

D_IO_ERROR
: Hardware IO error

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

D_OUT_OF_BAND
: Out-of-band condition occurred on device (such as typing control-C)

D_READ_ONLY
: Data cannot be written to this device.

D_WOULD_BLOCK
: Operation would block, but D_NOWAIT set

bytes_written
: [out scalar, in for asynchronous form] Size of data transfer.

RETURN VALUE

device_write_request returns only message transmission errors. The return value supplied to **ds_device_write_reply** is irrelevant. The return value from **device_write** (or *return_code*) can be a device error code if that error caused the operation to fail completely.

RELATED INFORMATION

Functions: **device_write_inband**, **device_reply_server**.

Status:

New: [OSF-RI MK6] Added to support asynchronous I/O completion from interrupt context.

device_write_async

Function — Write a sequence of bytes to a device object.

LIBRARY

#include <**device/device.h**> (**device_write_async**)

SYNOPSIS

kern_return_t **device_write_async**
(device_port_t *device,*
mach_port_t *queue;*
mach_port_t *request_id;*
dev_mode_t *mode,*
recnum_t *recnum,*
io_buf_ptr_t *data,*
io_buf_len_t *bytes_wanted)*;

DESCRIPTION

The **device_write_async** function enqueues a write operation for a sequence of bytes to a device object. The meaning of *recnum* as well as the specific operation performed is device dependent.

PARAMETERS

device
[in device send right] A device port to the device to be read.

queue
[in io_done queue send right] The port returned from **io_done_-queue_create**.

request_id
[in send right] An unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
[in scalar] I/O mode value. Meaningful options are:

D_NOWAIT
Do not wait if data is unavailable.

recnum
[in scalar] Record number to be read.

data
: [pointer to in array of bytes] Data bytes to be written.

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_write_async returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async_inband, device_read_overwrite_async, device_write_async_inband, io_done_queue_create**.

Status:

Believed to be stable

Use of the I/O operations limit is experimental.
[OSF-RI DIPC]

device_write_inband

Function — Write a sequence of bytes “inband” to a device object.

LIBRARY

#include **<device/device.h>** (**device_write_inband**)

#include **<device/device_request.h>** (**device_write_request_inband**)

#include **<device/device_reply.h>** (**ds_device_write_reply_inband**)

SYNOPSIS

kern_return_t **device_write_inband**

(device_port_t	*device,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
pointer_t	*data,*
mach_msg_type_number_t	*data_count,*
kern_return_t*	*device_code*,
io_buf_len_t*	*bytes_written*);

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

device_write_request_inband

Asynchronous Function form — Asynchronously write data

kern_return_t **device_write_request_inband**

(device_port_t	*device,*
mach_port_t	*reply_port,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
pointer_t	*data,*
mach_msg_type_number_t	*data_count*);

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

ds_device_write_reply_inband

Asynchronous Server Interface form — Receive the reply from an asynchronous write

kern_return_t **ds_device_write_reply_inband**

(mach_port_t	*reply_port,*
kern_return_t	*return_code,*
kern_return_t	*device_code,*
io_buf_len_t	*bytes_written*);

DESCRIPTION

The **device_write_inband** function writes a sequence of bytes to a device object. The meaning of *recnum* as well as the specific operation performed is device dependent. This call differs from **device_write** in that the bytes to be written are sent “inband” in the request IPC message.

The **device_write_inband** function will block until the current number of operations on the device is less than the device's I/O operations limit.

PARAMETERS

device
: [in device send right] A device port to the device to be written.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which the reply message is to be sent.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait for I/O completion.

recnum
: [in scalar] Record number to be written.

data
: [pointer to in array of bytes] Data bytes to be written.

data_count
: [in scalar] Number of data bytes to be written.

return_code
: [in scalar] The return status code from the write.

device_code
: [out scalar, in for asynchronous form] Device completion status. If the return value from the call (*return_code* in the asynchronous case) is not KERN_SUCCESS, the operation failed outright. If the operation partially succeeded, the return value is KERN_SUCCESS and *device_-code* reflects the status of the partial completion, D_SUCCESS if the operation was completely successful.

  D_DEVICE_DOWN
  : Device has been shut down

  D_INVALID_RECNUM
  : Invalid record (block) number

  D_INVALID_SIZE
  : Invalid IO size

  D_IO_ERROR
  : Hardware IO error

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.

D_OUT_OF_BAND
Out-of-band condition occurred on device (such as typing control-C)

D_READ_ONLY
Data cannot be written to this device.

D_WOULD_BLOCK
Operation would block, but D_NOWAIT set

bytes_written
[out scalar, in for asynchronous form] Size of data transfer.

RETURN VALUE

device_write_request_inband returns only message transmission errors. The return value supplied to **ds_device_write_reply_inband** is irrelevant. The return value from **device_write_inband** (or *return_code*) can be a device error code if that error caused the operation to fail completely.

RELATED INFORMATION

Functions: **device_write**, **device_reply_server**.

device_write_async_inband

Status:

New: [OSF-RI MK6] Added to support asynchronous I/O completion from interrupt context.

Function — Write a sequence of bytes “inband” to a device object.

LIBRARY

#include **<device/device.h>** (**device_write_async_inband**)

SYNOPSIS

kern_return_t **device_write_async_inband**

(device_port_t	*device,*
mach_port_t	*queue;*
mach_port_t	*request_id;*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_ptr_t	*data,*
io_buf_len_t	*bytes_wanted)*;

DESCRIPTION

The **device_write_async_inband** function enqueues a write operation for a sequence of bytes to a device object. The meaning of *recnum* as well as the specific operation performed is device dependent. This call differs from **device_write_async** in that the bytes to be written are sent “inband” in the request IPC message.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from **io_done_-queue_create**.

request_id
: [in send right] An unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

data
: [pointer to in array of bytes] Data bytes to be written.

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_write_async_inband returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async_inband, device_read_overwrite_async, device_write_async, io_done_queue_create**.

io_done_queue_create

Function — Create an io_done_queue kernel object

Status:

New: [OSF-RI MK6] Function added to support asynchronous I/O from interrupt context.

SYNOPSIS

```
kern_return_t io_done_queue_create
                (host_name_port_t          host,
                 mach_port_t*              queue);
```

DESCRIPTION

The **io_done_queue_create** function is called to create a new instatiation of the kernel object supporting asynchronous read/write operations on a device.

PARAMETERS

host
: [in host-name send right] The name (or control) port for the host on which the io_done_queue should be created.

queue
: [out io-done-queue send right] The port referencing the created io_done_queue.

RETURN VALUE

???

RELATED INFORMATION

Functions: **io_done_queue_terminate**, **io_done_queue_wait**, **device_read_async**, **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**.

io_done_queue_terminate

Status:

New: [OSF-RI MK6] Function added to support asynchronous I/O from interrupt context.

Function — Terminate an io_done_queue kernel object

SYNOPSIS

kern_return_t **io_done_queue_terminate**
(mach_port_t* *queue*);

DESCRIPTION

The **io_done_queue_terminate** function is called to destroy a previous instatiation of the kernel object supporting asynchronous read/write operations on a device.

PARAMETERS

queue
[in io-done-queue send right] The port referencing the io_done_queue to be destroyed.

RETURN VALUE

???

RELATED INFORMATION

Functions: **io_done_queue_create**, **io_done_queue_wait**, **device_read_async**, **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**.

io_done_queue_wait

Function — Wait on an io_done_queue kernel object

Status:

New: [OSF-RI MK6] Function added to support asynchronous I/O from interrupt context.

SYNOPSIS

```
kern_return_t io_done_queue_wait
        (mach_port_t*                    queue
         io_done_result_t*               result);
```

DESCRIPTION

The **io_done_queue_wait** function is called to obtain the results of a previously requested asynchronous I/O operation. For each **io_done_queue_wait** invocation, the status of one I/O request is returned. If there are no pending I/O completions, **io_done_queue_wait** blocks in the kernel on the address of the completion queue. The µKernel, from interrupt context, enqueues (in FIFO order) completions (struct io_done_result's) on the completion queue and posts a wakeup on the queue for each I/O completion. Completion processing previously done by the µKernel io_done thread is now done by the task thread when it awakens.

PARAMETERS

queue
: [in io-done-queue send right] The port referencing the io_done_queue to be destroyed.

result
: [out structure] The data structure to be filled in with the completion status of the I/O operation.

RETURN VALUE

???

RELATED INFORMATION

Functions: **io_done_queue_create**, **io_done_queue_wait**, **device_read_async**, **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**.

Data Structures: **io_done_result**.

Status:

Believed to be stable

receive_packet

Server Interface — Handles the arrival of a device packet

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **receive_packet**
(filter_port_t *filter_port,*
char* *header*,
char* *packet*);

DESCRIPTION

A **receive_packet** function is called by **device_server** as the result of a kernel message indicating that a packet meeting the criteria of a device's packet filter has been received.

Note that there are additional, related, interfaces that are generated by MIG and documented in the server writer's guide(e.g. maximum size of a received message, how to invoke receive unmarshalling stub, etc.)

PARAMETERS

filter_port
: [in filter (receive) right] The port to which the packet message was sent.

header
: [pointer to in array of char] The packet header, of size NET_HDW_HDR_MAX (declared in **<device/device.h>**).

packet
: [pointer to in array of char] The packet body, of size NET_RCV_MAX (declared in **<device/device.h>**).

NOTES

The kernel implementation of a network device is free to drop packets. For example, a network device will drop a message (i.e. a packet) if the attempt to send the message encounters a full queue. This is necessary to avoid flooding receivers with packets.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **device_set_filter**, **device_reply_server.**

APPENDIX A Network Data Representation

The definition of Mach IPC permits for arbitrary encoding of the message data values into the message body and out-of-line areas. Direct users of the **mach_msg** service are free to encode data as desired.

The MIG stub generator encodes data values according to the Network Data Representation format (NDR). Since the kernel is a recipient of messages and contains MIG stubs to decode them (and to encode messages and replies it itself sends), NDR can be considered to be the kernel's data encoding. This encoding is described in this appendix.

Data Format Overview

Data generated and interpreted by MIG stubs is encoded in the message body and possibly out-of-line regions. These data areas can be viewed as a single data stream with the out-of-line region contents inserted into the message body data at the appropriate points.

The data stream begins with a format label whose format is machine independent which can be used to determine format conversions to be performed on the decoded data. This label doesn't actually describe the types of the data in the message; it simply says what machine format was used for the values that are encoded.

Following the format label are the encoded data values. Scalar data values are always directly placed into the data stream with no conversion (it is the receiver's responsibility to transform the formats if necessary). Structured values are ordered according to a prescribed set of machine independent rules, sometimes with header information describing the value's size, for example.

Out-of-line memory is used to transmit large data areas. The order of these out-of-line regions in the message is dictated by the MIG stub. No additional information is present in the data stream to describe these regions (since arrays or structures of out-of-line regions are not currently supported by MIG).

The order of port rights in the message body are also dictated by the MIG stub. For scalar rights, no additional information appears in the data stream. For port arrays, however, the information concerning the size of the array (if variable) is encoded in the data stream just as if the data were to appear there.

The Network Data Representation format specification is capable of representing more complicated types than MIG can describe. This appendix describes only that subset of NDR necessary to describe MIG types.

Data Representation Format Label

The first eight bytes of the data stream is the data representation format label used by the receiver to convert the scalar formats in the stream from the sender's representation into that of the receiver.

```
struct NDR_record_t
{
        unsigned char           mig_vers;
        unsigned char           if_vers;
        unsigned char           reserved1;
        unsigned char           mig_encoding;
        unsigned char           int_rep;
        unsigned char           char_rep;
        unsigned char           float_rep;
        unsigned char           reserved2;
};
```

mig_vers
Currently zero.

if_vers
Currently zero.

mig_encoding
Format of MIG encoding, currently NDR_PROTOCOL_2_0.

int_rep
The way in which the sender orders the bytes in an integer value.

NDR_INT_BIG_ENDIAN
Big endian

NDR_INT_LITTLE_ENDIAN
Little endian

char_rep
The major encoding of single byte characters.

NDR_CHAR_ASCII
ASCII

NDR_CHAR_EBCDIC
EBCDIC

float_rep
The format family used by the sender for floating point values.

NDR_FLOAT_IEEE
IEEE (single and double format)

NDR_FLOAT_VAX
VAX (F and G format)

NDR_FLOAT_CRAY
Cray (IEEE single and Cray double format)

NDR_FLOAT_IBM
IBM (short and long format)

Scalar Type Format

A scalar value is a single value as opposed to a structure or array value. The following sections describe the way in which the scalar values that compose structure or array types are ordered in the data stream. In one sense, the data stream consists of a stream of scalar values, some of which are used to describe other values in the stream.

All scalar values are transmitted in the format used by the message sender. All values must be naturally aligned: a value *N* bytes wide must start at a 0 mod *N* byte stream index (considering the first byte in the stream to be index 0). If necessary, (the smallest number of) additional bytes are added to the stream (of arbitrary value) to preserve this alignment (unless the next value would be out-of-line, in which case the out-of-line region must have the proper alignment).

Boolean

The MIG fundamental type MACH_MSG_TYPE_BOOLEAN represents a single boolean value. This value is encoded as a single byte:

- 0—False
- non-zero—True

Character

The MIG fundamental type MACH_MSG_TYPE_CHAR represents a single byte character of the format shown by the format label. Each character consumes a single byte in the data stream.

Integer

The MIG fundamental types MACH_MSG_TYPE_INTEGER_8, MACH_MSG_TYPE_INTEGER_16, MACH_MSG_TYPE_INTEGER_32 and MACH_MSG_TYPE_INTEGER_64 represent an integer of the specified length (in bits for the MIG specification). These types are encoded as a 1, 2, 4 or 8 byte value, respectively, with the endianess shown in the format label.

Floating-Point

The MIG fundamental types MACH_MSG_TYPE_REAL_32 and MACH_MSG_TYPE_REAL_64 represent single or double precision floating point values. These values are encoded as four or eight bytes, respectively, using the format shown in the format label.

Un-interpreted Byte

The MIG fundamental type MACH_MSG_TYPE_BYTE represents a single un-interpreted byte. It is encoded in the data stream as a single byte. No format conversion takes place for this value.

Port Right

When a parameter to a MIG stub is a port right (as opposed to a port name), the right must be expressed in the port right descriptors in the message body. Port rights do not consume any bytes in the data stream.

Structure Format

Since MIG can only specify homogenous structures (with each element being a fixed size), the rule for structure encoding is simple: the structure is encoded as the set of structure members packed together in their declared order within the structure with the set at an alignment boundary appropriate for any member of the structure. If, however, the data elements of the structure are port rights, no data is encoded in the data stream; the (fixed number of) port rights will be described in the port right descriptor arrays.

Array Format

MIG arrays are a singly dimensioned set of a (possibly variable) number of fixed sized entities. These entities will be encoded in the data stream packed one after the other in index order (array element 0, then 1, etc.) at a data alignment appropriate for any given member.

For fixed sized arrays, only the data values will be encoded (unless they are port rights, in which case they will be described instead in the port right descriptor arrays).

Bounded Variable-Sized Arrays

MIG variable (but bounded) arrays are encoded as NDR "conformant" arrays. Prior to the first element of the array will be a four byte current length value for the array. If the elements of the array are port rights, the port rights will be described in the port right descriptor arrays but the count field will be encoded in the data stream at the point where the array would have appeared.

CountInOut Parameter

When sending a reply message containing a variable-sized array, it is desirable to send only as much as the requestor will accept. For this purpose, MIG has the *countinout* IPC flag. From the point of view of the data stream representation, this flag causes the request message to encode the requestor's count parameter as an additional data stream value. This extra value is not encoded in the reply data stream (just the array so bounded).

Fixed-Sized Strings

The MIG fixed-sized C string declaration is treated as the equivalent fixed-sized array of characters.

Variable-Sized Strings

The MIG variable-sized C string data is encoded using the NDR "noncorformant" string type. This representation consists of the offset of the sent string within the original string (always zero for MIG), the number of characters sent (including the terminator) followed by the characters themselves.

Co-Bounded Variable-Sized Arrays

If several (adjacent) MIG parameters are variable-sized arrays known to be of the same extent (*samecount* keyword), the first array is encoded as a bounded variable-sized array and the others are encoded as fixed-sized arrays (of the size of the first array).

Out-of-Line Format

For each "pointer" type parameter in the MIG specification, a distinct out-of-line region will be sent. The data so represented does not appear in the message body. The out-of-line region must have the alignment required by its contents. Any values necessary to describe the out-of-line region (array sizes) will be encoded into the message body portion of the data stream at the appropriate point.

Note that NDR “pointer” types (intended to transmit pointers) are not used for MIG “pointer” types (intended to specify out-of-line transmission).

APPENDIX B

MIG Server Routines

This appendix describes server message de-multiplexing routines generated by MIG from the kernel interface definitions of use to a server in handling messages sent from the kernel.

Each of these routines take two parameters: a pointer to a request message and a pointer to a reply message. The request message is in the format chosen by the sender of that message (i.e., not necessarily the local machine's format); the reply message must be in the format of the local machine.

Status:

Believed to be stable

clock_reply_server

Function — Handle kernel generated clock alarms

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t clock_reply_server
        (mach_msg_header_t*        in_msg,
         mach_msg_header_t*        out_msg);
```

DESCRIPTION

The **clock_reply_server** function is the MIG generated server handling function to handle messages from the kernel corresponding to clock alarms. Such messages are delivered to the alarm reply port named in a **clock_alarm** call. The **clock_reply_server** function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

PARAMETERS

in_msg
: [pointer to in structure] The alarm message received from the kernel.

out_msg
: [out structure] Not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to the alarm mechanism and no other action was taken.

RELATED INFORMATION

Functions: **clock_alarm_reply**.

device_reply_server

Status:

Believed to be stable

Function — Handles messages from a kernel device driver

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

boolean_t **device_reply_server**
(mach_msg_header_t* *in_msg*,
mach_msg_header_t* *out_msg*);

DESCRIPTION

The **device_reply_server** function is the MIG generated server handling function to handle messages from kernel device drivers. Such messages were sent in response to the various **device_..._request**... calls. It is assumed when using those calls that some task is listening for reply messages on the port named as a reply port to those calls. The **device_reply_server** function performs all necessary argument handling for a kernel message and calls one of the device server functions to interpret the message.

PARAMETERS

in_msg
: [pointer to in structure] The device driver message received from the kernel.

out_msg
: [out structure] A reply message. No messages from a device driver expect a direct reply, so this field is not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to this device handler interface and no other action was taken.

RELATED INFORMATION

Functions: **ds_device_open_reply**, **ds_device_write_reply**, **ds_device_write_reply_inband**, **ds_device_read_reply**, **ds_device_read_reply_inband**, **ds_device_read_reply_overwrite**, **receive_packet**.

exc_server

Status:

Believed to be stable

Function — Handles kernel messages for an exception handler

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

boolean_t **exc_server**
(mach_msg_header_t* *in_msg,*
mach_msg_header_t* *out_msg*);

DESCRIPTION

The **exc_server** function is the MIG generated server handling function to handle messages from the kernel relating to the occurrence of an exception in a thread. Such messages are delivered to the exception port set via **thread_set_exception_port** or **task_set_exception_port**. When an exception occurs in a thread, the thread sends an exception message to its exception port, blocking in the kernel waiting for the receipt of a reply. The **exc_server** function performs all necessary argument handling for this kernel message and calls **catch_exception_raise**, **catch_exception_raise_state** or **catch_exception_raise_state_identity**, which should handle the exception. If the called routine returns KERN_SUCCESS, a reply message will be sent, allowing the thread to continue from the point of the exception; otherwise, no reply message is sent and the called routine must have dealt with the exception thread directly.

PARAMETERS

in_msg
[pointer to in structure] The exception message received from the kernel.

out_msg
[out structure] A reply message.

RETURN VALUE

TRUE
The message was handled and the appropriate function was called.

FALSE

The message did not apply to the exception mechanism and no other action was taken.

RELATED INFORMATION

Functions: **catch_exception_raise**.

memory_object_default_server

Function — Handles kernel messages for the default memory manager

Status:

Believed to be stable

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

boolean_t **memory_object_default_server**
(mach_msg_header_t* *in_msg*,
mach_msg_header_t* *out_msg*);

DESCRIPTION

The **memory_object_default_server** function is the MIG generated server handling function to handle messages from the kernel targeted to the default memory manager. This server function only handles messages unique to the default memory manager. Messages that are common to all memory managers are handled by **memory_object_server**.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **memory_object_default_server** function performs all necessary argument handling for a kernel message and calls one of the default memory manager functions.

PARAMETERS

in_msg
: [pointer to in structure] The memory manager message received from the kernel.

out_msg
: [out structure] A reply message. Note that no kernel messages to a memory manager expect a direct reply.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE

The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **seqnos_memory_object_default_server**, **memory_object_server**, **memory_object_create**, **memory_object_data_initialize**, **default_pager_object_create**, **default_pager_info**, **default_pager_paging_file**.

memory_object_server

Status:

Believed to be stable

Function — Handles kernel messages for a memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t memory_object_server
                (mach_msg_header_t*                    in_msg,
                 mach_msg_header_t*                    out_msg);
```

DESCRIPTION

The **memory_object_server** function is the MIG generated server handling function to handle messages from the kernel targeted to a memory manager.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **memory_object_server** function performs all necessary argument handling for a kernel message and calls one of the memory manager functions to interpret the message.

PARAMETERS

in_msg
: [pointer to in structure] The memory manager message received from the kernel.

out_msg
: [out structure] A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **memory_object_default_server**, **memory_object_data_request**, **memory_object_data_return**, **memory_object_data_unlock**, **memory_object_notify**, **memory_object_lock_completed**, **memory_object_change_completed**, **memory_object_supply_completed**, **memory_object_terminate**, **memory_object_rejected**, **memory_object_synchronize**, **memory_object_force_copy_completed**, **seqnos_memory_object_server**.

notify_server

Status:

Believed to be stable

Function — Handle kernel generated IPC notifications

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

boolean_t **notify_server**
(mach_msg_header_t* *in_msg*,
mach_msg_header_t* *out_msg*);

DESCRIPTION

The **notify_server** function is the MIG generated server handling function to handle messages from the kernel corresponding to IPC notifications. Such messages are delivered to the notification port named in a **mach_msg** or **mach_port_request_notification** call. The **notify_server** function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

PARAMETERS

in_msg
[pointer to in structure] The notification message received from the kernel.

out_msg
[out structure] Not used.

RETURN VALUE

TRUE
The message was handled and the appropriate function was called.

FALSE
The message did not apply to the notification mechanism and no other action was taken.

RELATED INFORMATION

Functions: **seqnos_notify_server**, **do_mach_notify_dead_name**, **do_mach_-notify_no_senders**, **do_mach_notify_port_deleted**, **do_mach_notify_send_once**.

prof_server

Status: **Believed to be stable**

Function — Handle kernel generated PC sample messages

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t prof_server
        (mach_msg_header_t*            in_msg,
         mach_msg_header_t*            out_msg);
```

DESCRIPTION

The **prof_server** function is the MIG generated server handling function to handle messages from the kernel corresponding to program counter (profiling) samples. Such messages are delivered to the task or thread sample port set by **task_sample** or **thread_sample**. The **prof_server** function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

PARAMETERS

in_msg
: [pointer to in structure] The sample message received from the kernel.

out_msg
: [out structure] Not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to the sample mechanism and no other action was taken.

RELATED INFORMATION

Functions: **receive_samples**.

Status:

Believed to be stable

seqnos_memory_object_default_server

Function — Handles kernel messages for the default memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t seqnos_memory_object_default_server
        (mach_msg_header_t*                 in_msg,
         mach_msg_header_t*                 out_msg);
```

DESCRIPTION

The **seqnos_memory_object_default_server** function is the MIG generated server handling function to handle messages from the kernel targeted to the default memory manager. This server function only handles messages unique to the default memory manager. Messages that are common to all memory managers are handled by **seqnos_memory_object_server**.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **seqnos_memory_object_default_server** function performs all necessary argument handling for a kernel message and calls one of the default memory manager functions.

PARAMETERS

in_msg
: [pointer to in structure] The memory manager message received from the kernel.

out_msg
: [out structure] A reply message. Note that no kernel messages to a memory manager expect a direct reply.

NOTES

seqnos_memory_object_default_server differs from **memory_object_default_server** in that it supplies message sequence numbers to the server interfaces it calls.

RETURN VALUE

TRUE
The message was handled and the appropriate function was called.

FALSE
The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **memory_object_default_server**, **seqnos_memory_object_server**, **seqnos_memory_object_create**, **seqnos_memory_object_data_initialize**, **seqnos_default_pager_object_create**, **seqnos_default_pager_info**, **seqnos_default_pager_paging_file**.

seqnos_memory_object_server

Status:

Believed to be stable

Function — Handles kernel messages for a memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

boolean_t **seqnos_memory_object_server**
(mach_msg_header_t* *in_msg*,
mach_msg_header_t* *out_msg*);

DESCRIPTION

The **seqnos_memory_object_server** function is the MIG generated server handling function to handle messages from the kernel targeted to a memory manager.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **seqnos_memory_object_server** function performs all necessary argument handling for a kernel message and calls one of the memory manager functions to interpret the message.

PARAMETERS

in_msg
[pointer to in structure] The memory manager message received from the kernel.

out_msg
[out structure] A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

NOTES

seqnos_memory_object_server differs from **memory_object_server** in that it supplies message sequence numbers to the server interfaces.

RETURN VALUE

TRUE
The message was handled and the appropriate function was called.

FALSE
: The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **seqnos_memory_object_default_server**, **seqnos_memory_object_data_request**, **seqnos_memory_object_data_unlock**, **seqnos_memory_object_data_return**, **seqnos_memory_object_supply_completed**, **seqnos_memory_object_notify**, **seqnos_memory_object_lock_completed**, **seqnos_seqnos_memory_object_change_completed**, **seqnos_memory_object_terminate**, **seqnos_memory_object_synchronize**, **seqnos_memory_object_rejected**, **seqnos_memory_object_force_copy_completed**, **memory_object_server**.

Status:

Believed to be stable

seqnos_notify_server

Function — Handle kernel generated IPC notifications

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t seqnos_notify_server
        (mach_msg_header_t*        in_msg,
         mach_msg_header_t*        out_msg);
```

DESCRIPTION

The **seqnos_notify_server** function is the MIG generated server handling function to handle messages from the kernel corresponding to IPC notifications. Such messages are delivered to the notification port named in a **mach_msg** or **mach_port_request_notification** call. The **seqnos_notify_server** function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

PARAMETERS

in_msg
: [pointer to in structure] The notification message received from the kernel.

out_msg
: [out structure] Not used.

NOTES

seqnos_notify_server differs from **notify_server** in that it supplies message sequence numbers to the server interfaces.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to the notification mechanism and no other action was taken.

RELATED INFORMATION

Functions: **notify_server**, **do_seqnos_mach_notify_dead_name**, **do_seqnos_mach_notify_no_senders**, **do_seqnos_mach_notify_port_deleted**, **do_seqnos_mach_notify_send_once**.

APPENDIX C

Default Memory Management Interface

In general, the default memory manager is just like any other memory manager, except that it is “trusted” to respond promptly to paging requests in as much as that it is the memory manager of last resort. There are a few special requests issued to the default memory manager having to do with the creation and management of anonymous memory.

The following interfaces will likely be separated into a different document in a future release of the specification:

- default_pager_add_segment
- default_pager_backing_store_create
- default_pager_backing_store_delete
- default_pager_backing_store_info
- default_pager_objects
- default_pager_object_pages

Status:

New – expanded DMMI suite. **[OSF/1 R1.3]**

default_pager_add_segment

Server Interface —Add additional backing storage for a default pager

LIBRARY

libmach.a only

#include **<mach/default_pager_object.h>**

SYNOPSIS

```
kern_return_t default_pager_add_segment
        (mach_port_t          backing_store,
         mach_port_t          device,
         recnum_t             offset,
         recnum_t             count,
         integer_t            record_size);
```

DESCRIPTION

The **default_pager_add_segment** function is called to add a partition to a default pager's backing storage (i.e. expand the amount of backing storage available to a memory manager). The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by tasks holding the backing store port, created with **default_pager_backing_store_create**, for a default memory manager.

The result is that the pager may use *count* records on *device* starting at *offset* for paging, and each record is *record_size* bytes in length (note that the **device_*** calls are, or can be, record oriented).

PARAMETERS

backing_store
: [in backing store (receive) right] The backing store port.

device
: [in device port] The port for the device containing the backing storage partition.

offset
: [in scalar] The offset, in *record_size* units, to the beginning of the backing storage on the device.

count
: [in scalar] The number of units ($\log_2$ of `vm_page_size`/*record_size*) in the partition/segment.

record_size
[in scalar] The size, in bytes, of the storage device record.

RETURN VALUE

KERN_FAILURE
The default pager does not support this operation.

KERN_INVALID_ARGUMENT
The *backing_store* port does not represent a valid backing store or the specified segment overlaps an existing partition.

KERN_SUCCESS
The operation was successful.

RELATED INFORMATION

Functions: **default_pager_backing_store_create**, **default_pager_backing_store_delete**, **default_pager_backing_store_info**.

Status:

New – expanded DMMI suite. **[OSF/1 R1.3]**

default_pager_backing_store_create

Server Interface — Create a backing storage object

LIBRARY

libmach.a only

#include **<mach/default_pager_object.h>**

SYNOPSIS

```
kern_return_t default_pager_backing_store_create
                (mach_port_t                pager,
                 integer_t                  priority,
                 integer_t                  clsize
                 mach_port_t*               backing_store);
```

DESCRIPTION

The **default_pager_backing_store_create** function is called to create a new backing storage object. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by tasks (privileged) holding the default pager service port.

PARAMETERS

pager
: [in default pager (receive) right] The default pager service port.

priority
: [in scalar] The scheduling priority for the backing store service thread(s).

clsize
: [in scalar] The preferred cluster size (in bytes) for the backing store object.

backing_store
: [out backing store (receive) right] The port used to manipulate the created backing store.

RETURN VALUE

KERN_FAILURE
: The default pager does not support this operation.

KERN_INVALID_ARGUMENT
The *pager* port does not represent a valid default pager.

KERN_SUCCESS
The operation was successful.

RELATED INFORMATION

Functions: **default_pager_backing_store_add_segment**, **default_pager_backing_store_delete**, **default_pager_backing_store_info**.

Status:

New – expanded DMMI suite. **[OSF/1 R1.3]**

default_pager_backing_store_delete

Server Interface — Delete a backing storage object

LIBRARY

libmach.a only

#include <**mach/default_pager_object.h**>

SYNOPSIS

kern_return_t **default_pager_backing_store_delete**
(mach_port_t *backing_store*);

DESCRIPTION

The **default_pager_backing_store_delete** function is called to destroy a backing storage object created by **default_pager_backing_store_create**. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by tasks holding the backing store port, created with **default_pager_backing_store_create**, for a default memory manager.

PARAMETERS

backing_store
: [in backing store (receive) right] The backing store port created by **default_pager_backing_store_create**.

RETURN VALUE

KERN_FAILURE
: The default pager does not support this operation.

KERN_INVALID_ARGUMENT
: The *backing_store* port does not represent a valid backing store or the specified segment overlaps an existing partition.

KERN_SUCCESS
: The operation was successful.

RELATED INFORMATION

Functions: **default_pager_backing_store_add_segment**, **default_pager_backing_store_create**, **default_pager_backing_store_info**.

default_pager_backing_store_info

Status:

New – expanded DMMI suite. **[OSF/1 R1.3]**

Server Interface — Return information about a backing storage object

LIBRARY

libmach.a only

#include **<default_pager/mach/default_pager_types.h>**

SYNOPSIS

kern_return_t **default_pager_backing_store_info**

(mach_port_t	*backing_store*
backing_store_flavour_t	*flavor*,
backing_store_info_t	*info*,
mach_msg_type_number_t*	*size*);

DESCRIPTION

The **default_pager_backing_store_info** function is called to obtain information about a backing storage object created by **default_pager_backing_store_create**. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by tasks holding the backing store port, created with **default_pager_backing_store_create**, for a default memory manager.

PARAMETERS

backing_store
: [in backing store (receive) right] The backing store port for which information is desired.

flavor
: [in scalar] The type of information to be changed. Valid values are:

  BACKING_STORE_BASIC_INFO
  : Statistical and space used/available information. It includes the priority and cluster size that was provided in the **default_pager_backing_store_create** call.

info
: [pointer to in structure] The data structure that will be filled-in with the information provided for the requested *flavor*.

size
: [pointer to in/out scalar] On input, the maximum size of the *info* data structure; on output, the actual size of the returned data.

RETURN VALUE

KERN_FAILURE
: The default pager does not support this operation.

KERN_INVALID_ARGUMENT
: The *backing_store* port does not represent a valid backing store, *flavor* is not valid, or *size* is not the size for the requested flavor..

KERN_SUCCESS
: The operation was successful.

RELATED INFORMATION

Functions: **default_pager_backing_store_add_segment**, **default_pager_backing_store_create**, **default_pager_backing_store_delete**.

Data Structures: **backing_store_basic_info**.

default_pager_info

Status:

Believed to be stable

Server Interface —Return backing storage information

LIBRARY

libmach.a only

#include **<mach/default_pager_object.h>**

SYNOPSIS

```
kern_return_t default_pager_info
                (default_pager_port_t                    pager,
                 default_pager_info_t*                   info);
```

seqnos_default_pager_info

Sequence Number form

```
kern_return_t seqnos_default_pager_info
                (default_pager_port_t                    pager,
                 mach_port_seqno_t                       seqno,
                 default_pager_info_t*                   info);
```

DESCRIPTION

A **default_pager_info** function is called as the result of a message requesting that the default memory manager return information concerning the default pager's paging partitions. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding the default memory manager port.

PARAMETERS

pager
: [in default-pager (receive) right] The default memory manager service port.

seqno
: [in scalar] The sequence number of this message relative to the pager port.

info
: [out structure] Total and free space consumption.

RETURN VALUE

The default memory manager should return KERN_SUCCESS if it returns the desired information and KERN_FAILURE if it does not support the operation.

RELATED INFORMATION

Functions: **vm_set_default_memory_manager**, **memory_object_default_-server**, **seqnos_memory_object_default_server**.

Data Structures: **default_pager_info**.

default_pager_object_create

Status:

Believed to be stable.

Poor (but privileged) man's shared memory

Server Interface — Create a memory object managed by the default pager

LIBRARY

#include **<mach/default_pager_object.h>**

SYNOPSIS

```
kern_return_t default_pager_object_create
                (default_pager_port_t                pager,
                memory_object_t*                     memory_object,
                vm_size_t                            object_size);
```

seqnos_default_pager_object_create

Sequence Number form

```
kern_return_t seqnos_default_pager_object_create
                (default_pager_port_t                pager,
                mach_port_seqno_t                    seqno,
                memory_object_t*                     memory_object,
                vm_size_t                            object_size);
```

DESCRIPTION

A **default_pager_object_create** function is called as the result of a message requesting that the default memory manager create and return a (shared) memory object which is suitable for use with **vm_map**. This memory object has the same properties as does a memory object provided by **vm_allocate**: its initial contents are zero and the backing contents are temporary in that they do not persist after the memory object is destroyed. The memory object is suitable for use as non-permanent shared memory. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding the default memory manager port. This call should be contrasted with the kernel's **memory_object_create** message, in which the memory cache object is already created and the identity of the abstract memory object is made known to the default manager.

PARAMETERS

pager
: [in default-pager (receive) right] The default memory manager service port.

seqno
: [in scalar] The sequence number of this message relative to the *pager* port.

memory_object
: [out memory-object-representative send right] A memory object representative port (with full access) for the memory object.

object_size
: [in scalar] The maximum size for the memory object.

RETURN VALUE

Return KERN_SUCCESS if the object was created.

RELATED INFORMATION

Functions: **vm_map**, **vm_set_default_memory_manager**, **memory_object_create**, **memory_object_default_server**, **seqnos_memory_object_default_server**.

default_pager_objects

Status:

New – expanded DMMI suite. **[OSF/1 R1.3]**

Server Interface — Return information about memory objects managed by a default pager

LIBRARY

libmach.a only

#include <**default_pager/mach/default_pager_types.h**>

SYNOPSIS

kern_return_t **default_pager_objects**
(mach_port_t *pager,*
default_pager_object_array_t* *objectsp*,
mach_msg_type_number_t* *ocount*,
mach_port_array_t* *portsp*,
mach_msg_type_number_t* *pcount*);

DESCRIPTION

The **default_pager_objects** function is called to obtain information about the objects managed by a default pager. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding the default memory manager port.

PARAMETERS

pager

[in backing store (receive) right] The backing store port for which information is desired.

objectsp

[out array of type default_pager_object_t] An array of memory objects managed by the default pager. Each element of the array has the following format:

```
struct default_pager_object {
        vm_offset_t             dpo_object;
        vm_size_t               dpo_size;
}
```

The `dpo_offset` describes the object managed by the pager; the `dpo_size` describes the backing store used for the object.

ocount
: [pointer to in/out scalar] On input, points to the size of *objectsp* in default_pager_object units; on output, indicates the number of valid entries in the array.

portsp
: [out array of mach_port_t] An array of ports for the memory objects managed by the default pager. The offset of each port is the same as the offset for the complimentary information in the *objectsp* array.

pcount
: [pointer to in/out scalar] On input, indicates the size of *portsp* in mach_port_t units; on output, indicates the number of valid entries in the array.

RETURN VALUE

KERN_FAILURE
: The default pager does not support this operation.

KERN_INVALID_ARGUMENT
: The *pager* port does not represent a valid default pager.

KERN_RESOURCE_SHORTAGE
: A kernel memory allocation error occured while performing the operation.

KERN_SUCCESS
: The operation was successful.

RELATED INFORMATION

Functions: **default_pager_object_pages**.

default_pager_object_pages

Server Interface — Return information about memory objects managed by a default pager

Status:

New – expanded DMMI suite. **[OSF/1 R1.3]**

LIBRARY

libmach.a only

#include <**default_pager/mach/default_pager_types.h**>

SYNOPSIS

```
kern_return_t default_pager_object_pages
        (mach_port_t                      pager,
         abs_mem_obj_port_t               memory_object,
         default_pager_page_array_t*      pages
         integer_t*                       count);
```

DESCRIPTION

The **default_pager_object_pages** function is called to obtain information about the objects managed by a default pager. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding the default memory manager port.

PARAMETERS

pager
: [in default-pager receive right] The default pager which manages the memory object for which information is desired.

memory_object
: [in abstract-memory-object send right] The memory object port to be examined. The port is the same as the ports returned by the **default_-pager_objects** call.

pages
: [out array of type default_pager_page_t] The array of page offsets comprising the memory object specified. Each element of the array has the following format:

```
struct default_pager_page {
        vm_offset_t                    dpp_offset;
}
```

The `dpp_offset` describes the offset of the page in its object.

count
: [out scalar] The number of entries in the *pages* array.

RETURN VALUE

KERN_FAILURE
: The default pager does not support this operation.

KERN_INVALID_ARGUMENT
: The *pager* port does not represent a valid default pager or the specified memory object was not found.

KERN_SUCCESS
: The operation was successful.

RELATED INFORMATION

Functions: **default_pager_objects**.

default_pager_paging_file

Status:

Obsolete – To Be Deleted
[OSF/1 R1.3]

Server Interface — Configure temporary backing storage

LIBRARY

libmach.a only

#include **<mach/default_pager_object.h>**

SYNOPSIS

```
kern_return_t default_pager_paging_file
        (default_pager_port_t                     pager,
         default_pager_filename_t                 file_name,
         boolean_t                                add);
```

seqnos_default_pager_paging_file

Sequence Number form

```
kern_return_t seqnos_default_pager_paging_file
        (default_pager_port_t                     pager,
         mach_port_seqno_t                        seqno,
         default_pager_filename_t                 file_name,
         boolean_t                                add);
```

DESCRIPTION

A **default_pager_paging_file** function is called as the result of a message requesting that the default memory manager change its backing storage configuration. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding the default memory manager port.

PARAMETERS

pager
: [in default-pager (receive) right] The default memory manager service port.

seqno
: [in scalar] The sequence number of this message relative to the pager port.

file_name
: [pointer to in array of *char*] A name of a paging partition or file.

add
: [in scalar] True if the specified paging space should be added to the manager's set of space; false if it should be removed.

RETURN VALUE

The default memory manager should return KERN_SUCCESS if it performs the desired change and KERN_FAILURE if it does not support the operation.

RELATED INFORMATION

Functions: **vm_set_default_memory_manager**, **memory_object_default_-server**, **seqnos_memory_object_default_server**.

host_set_default_memory_manager

Status:

New –
replaces vm_set_default_memory_manager.

Function — Sets the default memory manager.

SYNOPSIS

kern_return_t **host_set_default_memory_manager**

(host_control_port_t	*host_priv,*
default_pager_port_t*	*default_manager*,
vm_size_t	*cluster_size*);

DESCRIPTION

The **host_set_default_memory_manager** function establishes the default memory manager for a host. The named manager will be the target for future **memory_object_create** calls.

PARAMETERS

host_priv
: [in host-control send right] The control port naming the host for which the default memory manager is to be set.

default_manager
: [pointer to in/out default-pager send right] A memory manager port to the new default memory manager. If this value is MACH_PORT_NULL, the old memory manager is not changed. The old memory manager port is returned in this variable.

cluster_size
: [in scalar] The preferred cluster size (in bytes) for temporary memory objects.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **memory_object_create**, **vm_allocate**.

Status:

Believed to be stable

memory_object_create

Server Interface — Requests transfer of responsibility for a kernel-created memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **memory_object_create**

(default_pager_port_t	*pager,*
abs_mem_obj_port_t	*new_memory_object,*
vm_size_t	*new_object_size,*
mem_cache_control_port_t	*new_control,*
vm_size_t	*new_page_size*);

seqnos_memory_object_create

Sequence Number form

kern_return_t **seqnos_memory_object_create**

(default_pager_port_t	*pager,*
mach_port_seqno_t	*seqno*,
abs_mem_obj_port_t	*new_memory_object,*
vm_size_t	*new_object_size,*
mem_cache_control_port_t	*new_control,*
vm_size_t	*new_page_size*);

DESCRIPTION

A **memory_object_create** function is called as the result of a message from the kernel requesting that the default memory manager accept responsibility for the new memory object created by the kernel. The kernel makes this call only to the system default memory manager.

The new memory object initially consists of zero-filled pages. Only memory pages that are actually written are provided to the memory manager. When processing **memory_object_data_request** calls from the kernel, the default memory manager must use **memory_object_data_unavailable** for any pages that have not been written previously.

The kernel does not expect a reply to this call. The kernel assumes that the default memory manager will be ready to handle data requests to this object and does not need the confirmation of a **memory_object_establish** call.

PARAMETERS

pager
: [in default-pager (receive) right] The default memory manager service port.

seqno
: [in scalar] The sequence number of this message relative to the pager port.

new_memory_object
: [in abstract-memory-object receive right] The port representing the new abstract memory object created by the kernel. (Since there will be no **memory_object_notify / memory_object_establish** sequence, no memory object representative is created.)

new_object_size
: [in scalar] The expected size for the new object, in bytes.

new_control
: [in memory-cache-control send right] The memory cache control port to be used by the memory manager when making cache management requests for the new object.

new_page_size
: [in scalar] The page size used by the kernel. All calls involving this kernel must use data sizes that are integral multiples of this page size.

NOTES

The kernel requires memory objects to provide temporary backing storage for zero-filled memory created by **vm_allocate** calls, issued by both user tasks and the kernel itself. The kernel allocates an abstract memory object port to represent the temporary backing storage and uses **memory_object_create** to pass the new memory object to the default memory manager, which provides the storage.

The default memory manager is a trusted system component that is identified to the kernel at system initialization time. The default memory manager can also be changed at run time using the **vm_set_default_memory_manager** call.

The contents of a kernel-created (as opposed to a user-created) memory object can be modified only in main memory. The default memory manager must not change the contents of a temporary memory object, or allow unrelated tasks to access the memory object, control, or name port.

The kernel provides the size of a temporary memory object based on the allocated size. Since the object is not mapped by other tasks, the object will not grow by explicit action. However, the kernel may coalesce adjacent temporary objects in such a way that this object may appear to grow. As such, the supplied object size is merely a hint as to the maximum size.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY causes **mach_msg_server** to remove the abstract memory object and memory cache control port references.

RELATED INFORMATION

Functions: **default_pager_object_create**, **memory_object_data_initialize**, **memory_object_data_unavailable**, **memory_object_default_server**, **seqnos_memory_object_default_server**.

memory_object_data_initialize

Server Interface — Writes initial data back to a temporary memory object

Status:

Believed to be stable

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_data_initialize
        (abs_mem_obj_port_t                  memory_object,
         mem_cache_control_port_t            memory_control,
         vm_offset_t                         offset,
         pointer_t                           data,
         vm_size_t                           data_count);
```

seqnos_memory_object_data_initialize

Sequence Number form

```
kern_return_t seqnos_memory_object_data_initialize
        (abs_mem_obj_port_t                  memory_object,
         mach_port_seqno_t                   seqno,
         mem_cache_control_port_t            memory_control,
         vm_offset_t                         offset,
         pointer_t                           data,
         vm_size_t                           data_count);
```

DESCRIPTION

A **memory_object_data_initialize** function is called as the result of a kernel message providing the default memory manager with initial data for a kernel-created memory object. If the memory manager already has supplied data (by a previous **memory_object_data_initialize** or **memory_object_data_return**), it should ignore this call. Otherwise, the call behaves the same as the **memory_object_data_return** call.

The kernel makes this call only to the default memory manager and only on temporary memory objects that it has created with **memory_object_create**.

PARAMETERS

memory_object
: [in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied by the kernel in a **memory_object_create** call.

seqno
[in scalar] The sequence number of this message relative to the abstract memory object port.

memory_control
[in memory-cache-control send right] The memory cache control port to be used for a response by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

offset
[in scalar] The offset within the memory object.

data
[in pointer to dynamic array of bytes] The data that has been modified while cached in physical memory.

data_count
[in scalar] The number of bytes to be written, starting at *offset*. The number converts to an integral number of memory object pages.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY causes **mach_msg_server** to remove the memory cache control port reference and to de-allocate the returned data.

RELATED INFORMATION

Functions: **memory_object_create**, **memory_object_data_return**, **memory_object_default_server**, **seqnos_memory_object_default_server**.

vm_set_default_memory_manager

Status:

Obsolete – replaced by host_set_default_memory_manager.

Function — Sets the default memory manager.

SYNOPSIS

kern_return_t **vm_set_default_memory_manager**

(host_control_port_t	*host_priv,*
default_pager_port_t*	*default_manager*,
vm_size_t	*cluster_size*);

DESCRIPTION

The **vm_set_default_memory_manager** function establishes the default memory manager for a host. The named manager will be the target for future **memory_object_create** calls.

PARAMETERS

host_priv
: [in host-control send right] The control port naming the host for which the default memory manager is to be set.

default_manager
: [pointer to in/out default-pager send right] A memory manager port to the new default memory manager. If this value is MACH_PORT_NULL, the old memory manager is not changed. The old memory manager port is returned in this variable.

cluster_size
: [in scalar] The preferred cluster size (in bytes) for temporary memory objects.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **memory_object_create**, **vm_allocate**.